DOCKET NO.: MSFT-0736/183220.01
Application No.: 10/017,265
Office Action Dated: March 8, 2007

PATENT

Appendix A

A Framework for Next Generation Enterprise Application Integration

by

Andrew Roszko

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2004

Acknowledgements

My sincerest thanks are extended to my supervisor, Dr. Kostas Kontogiannis, not only for his encouragement, and financial support, but most of all for his genuine concern and interest on both professional and personal levels. His guidance has truly helped get the most out of this incredibly rewarding experience.

I'd also like to thank the Bell ExCite group for allowing me to be part of the EBCCS project; experiencing first hand the integration problems of a mission-critical, large scale system did indeed prove to be invaluable.

Thank you to the National Science and Engineering Research Council for their funding.

Finally, special thanks to the Electrical and Computer Engineering Department at the University of Waterloo, especially the admin staff, whose undying patience was indeed very much appreciated every step of the way.

Abstract

In addition to storing 70-75% of their data and business logic in legacy mainframe systems, global corporations have countless custom applications and off-the-shelf ERP products residing within their networks. Increasing competition and shrinking budgets have left managers scouring for innovative, cost-effective methods to maximize the potential of these enormous sunk costs. There is, as a result, an overwhelming need to not only web enable these existing legacy assets in order to quickly and cost-effectively deliver data to both customers and business partners alike, but also to amalgamate these disparate systems into a unified, homogeneous, real-time enterprise. Integration efforts to date, focused predominantly on the development of proprietary point-to-point adapters, have unfortunately proven to be a daunting task with countless failed projects and losses in the millions. The advent of XML web services does, however, have the potential to revolutionize existing integration strategies; the cost savings and ease of implementation associated with wrapping virtually all systems, past, present and future, with standardized, code-independent, data-centric interfaces is truly astounding. As the future success of this platform is, however, strictly dependent upon the interoperability of its endpoints, we have proposed several fundamental amendments to the existing flawed WSDL specification. A generic reference architecture, leveraging both this improved web services model as well as established component middleware technologies, is then proposed for the web enablement of legacy assets on an enterprise scale. In order to ensure the adoption of this methodology, a toolkit designed to automate the transformation has also been devised. This new paradigm will not only allow information to flow freely from deep within the enterprise, but will ultimately serve as the cornerstone of a new generation of enterprise integration solutions.

Table of Contents

Table of Figures

Chapter 1 Introduction

1.1 Motivation

It is estimated that while 'modern' legacy relics written in C, C++, and even Fortran make up a significant portion of the 500+ billion lines of reusable code currently owned by global corporations, 70% to 75% of this code base is actually stored in monolithic, centralized, host-based software systems [1, 2, 3]. An expenditure of 18 trillion dollars in the last decade alone on mergers and acquisitions, investment in superior technologies, as well as the purchase of third party CRM and ERP products has effectively left enterprises with truly heterogeneous applications and islands of information [4]. Even large, renowned organizations with exorbitant technology budgets are often unable to consolidate their data, thus degrading efficiency, driving up costs, and ultimately impacting customer satisfaction [5]. As the cost, risk and above all, development time associated with reengineering and/or redevelopment is far too significant, corporation survival lies in the ability to maximize the value of their existing assets. As a result, in addition to integrating these disparate systems, IT departments worldwide have been frantically devising architectures to deliver this legacy functionality over the internet.

In a market of shrinking budgets and margins, the integration of business applications into a single homogeneous environment is a compelling solution, for it essentially provides a unified face both internally as well as to customers and business partners. The creation of synergies between new and existing IT resources serve to procure a competitive edge, effectively increasing the speed with which a company can respond to changes in its business environment. Moreover, the ability of an enterprise to have access to real-time information spanning across multiple departments, applications, and platforms provides a plethora of far reaching benefits. The development of closer business relationships, improvement in supplier interactions, customer support and service, the creation of new business opportunities, increased revenues, decreased operating costs, and ultimately, greater market share in today's economy will all result from the implementation of a successful integration strategy [5, 6].

The advent of the web revolution has, however, dictated that the amalgamation of the enterprise is only part of a successful solution. The ease of use, low deployment costs, and ubiquity of the World Wide Web have created an overwhelming demand to expose the unified enterprise via HTTP. The mere concept of a thin client serves to save corporations millions in deployment and training costs alone. As a result, while a decade ago, developers were content pushing content to fat clients, they are now faced with the task of migrating both mainframe and client-server systems to the web. This methodology introduces the possibility of melting corporate barriers, thus allowing mission critical data to flow from deep within the enterprise directly to clients and/or business partners worldwide.

Though the need for a web-enabled, integrated enterprise is indeed apparent, the realization of such a platform is a daunting task. Milind Govekar, research director at the Gartner Group claims that the cost of the "glue" between applications is between 5 and 8 times greater than that of the applications themselves [7]. Many companies have attempted to leverage web application (CGI, ASP, JSP) and/or middleware (EJB, COM+) solutions to move data across the enterprise. However, in addition to relegating the entire information system to a tightly-coupled, vendor dependent environment, the need for costly, brittle proprietary adapters to wrap legacy applications with these technologies serves to drastically increase project risk. To this end, in a report studying the success of third party EAI solutions, Forrester Research found that on average, 45% of projects are completed late and over-budget [8]. This statistic is truly remarkable considering these platforms generally come packaged with a team of high priced, experts whose sole purpose is to coerce the system into working properly. Given the difficulty encountered in these resource rich endeavours, one can only imagine the struggle for success in the vast number of custom, in-house solutions. The daunting technological hurdles, expensive consultants, and countless failed projects with losses in the millions have indeed left companies weary of entering the integration arena. There is, however, the promise of a whole new interoperable world as XML web services slowly proliferate into the enterprise.

1.2 Thesis contributions

Though web services have garnered a tremendous amount of attention for their potential impact on inter enterprise integration, we explore the notion of pushing them from the periphery directly to the heart of the enterprise. Their standardized, platform neutral nature, coupled with a reliance on widely accepted protocols, offer a potential solution to the intra-enterprise integration woes outlined above. The promise of wrapping virtually any system with code-independent, data-centric interfaces has managers marvelling at the prospect of rapid, cost-effective implementations. The main objective of this paper is therefore to formalize an architecture to ensure that this dream does indeed come to fruition on the enterprise scale.

Stemming from a fundamental flaw in the excessively verbose WSDL 1.1 specification, the major drawback of web services to date has been the lack of interoperability between endpoints created with different toolsets. As this shortcoming clearly poses a threat to the success of the platform, we felt it was important to propose several key amendments to the spec. Having addressed the major deficiencies in the underlying plumbing, the web services platform, coupled with existing middleware technologies, are then leveraged to develop a reference architecture for the enterprise web enablement of legacy assets. Moreover, we attempt to ensure the rapid adoption of this approach with the proposal of an Enterprise Web Enablement (EWE) toolkit, which essentially automates the migration to the web. These contributions are then finally amalgamated into the creation of a methodology designed to transform intra-enterprise integration from a corporate nightmare to a mere implementation detail.

1.3 Thesis outline

Having presented the background information and related work in the second chapter, the third one addresses the interoperability issues prevalent in the web services paradigm; several key flaws in the WSDL spec as well as the measures required to address them are outlined. Chapter four then presents a reference architecture and toolkit for automating the process of legacy system web enablement, while the fifth chapter leverages this work to develop a methodology for drastically simplifying enterprise integration. Finally, chapter six provides conclusions as well as a discussion pertaining to future work.

Chapter 2
Background and Related Work

2.1 Modernization Strategies

Organizations worldwide have been dependent on information systems for the last two decades to efficiently run their operations and create new business opportunities; in essence, these systems are to an enterprise what a brain is to a higher species. As corporations evolve, requirements change and technologies providing a superior competitive advantage begin to emerge. This evolution, coupled with years of mergers and acquisitions, often leave organizations with hundreds of systems in desperate need of modernization and/or integration with new platforms. The heterogeneity of these applications can be rather astounding with transactional/batch processing systems on the mainframe, client-server applications written in a host of different languages and residing on differing platforms, as well as off-the-shelf ERP, CRM, ERM software all sitting within the same network. Managers are generally left with three main courses of action to modernize/unite these applications into a cohesive, efficient unit.

2.1.1 Redevelopment

The implementation of this alternative allows developers to start from scratch and redevelop all of the business applications without any consideration of the existing software. This rather daunting task can be carried out all at once, a technique dubbed the "big bang approach", or one application at a time, known as the "incremental approach" [9]. It can be noted that the only contact with the old systems in both cases is the migration of the data. This strategy does indeed appear attractive for it affords developers the opportunity to cleanly re-architect the applications based upon the most recent business and technology requirements. However, as every function must be redeveloped and tested in a new language running on a different platform, the completion of the entire development cycle will indeed be very costly and time consuming. Furthermore, the logic of many legacy systems is sufficiently complex that an attempt at duplication often results in products failing to meet the pre-defined requirements [10].

2.1.2 Reverse Engineering

Traditional reverse engineering emphasizes a deep understanding of individual modules followed by internal restructuring activities. The first step in most reengineering projects is program understanding, which has been defined as "the process of analyzing a subject system to identify the system's components and their interrelationships, and create representations of the system in another form or at a higher level of abstraction" [11, 12]. This process involves the modeling of the domain, the extraction of information from legacy systems using appropriate mechanisms, and creating abstractions that help in the understanding of the resulting structures [12]. The outcomes include redocumentation of both the architecture and the program structures as well as recovery of the design. Once the code has been analyzed and understood, restructuring can then take place. This methodology involves the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behaviour (functionality and semantics) [12]. This transformation is typically used to augment some quality attribute of the system, such as maintainability or performance.

Tools for program understanding, redocumentation, and program translation have indeed been applied successfully in the past; however, they depend inherently on the tractability of the starting legacy system. Old code is often so intertwined with its environment that its migration is virtually impossible without destroying it. Furthermore, it has been proven to be infeasible to convert procedural legacy programs to object oriented components; regardless of the success of this transformation, the responsible programmer will reject the results as the mental map of his/her algorithms has been destroyed [13]. Thus, as it does not take into account that programs arc a mirror of the programmers mind and cannot be reengineered without reengineering the mind of the programmer, classical reengineering in the sense of source code transformation, even via an intermediate language, is dead [14].

2.1.3 Wrapping

Wrapping or encapsulation, on the other hand, minimizes the need for deep, internal program understanding. Legacy systems are effectively wrapped in such a way that there is minimum change to the source; only the interfaces may be altered, so that the owner of the program is not estranged from it. A cocoon is essentially built around units of software that serve a well-defined need; this layer hides the unwanted complexity of the old system and exports a modern interface. Wrapping essentially serves to remove mismatches between the interface exported by a software artefact and the interfaces required by current integration practices [16]. It can be noted that over time, this methodology provides a means to replace modules individually without affecting the rest of the system.

It is evident that the selection of a modernization technique depends upon weighing the costs of strategy implementation as well as the benefits of the evolved system. In the technical dimension, one looks at the technical benefits of the improved software relative to the technical costs of employing the tools. In the business dimension, the benefits, including time to market and generated new opportunity are weighed against both the short and long term development and maintenance costs. Given the size and complexity of existing legacy systems, it is clear that there will always be significant risk, cost, and time associated with reengineering and redevelopment efforts. As a result, this paper aims to improve upon existing wrapping techniques in order to transform the process into a virtually routine procedure.

2.1.3.1 Techniques

A wide variety of software wrapping techniques have indeed been devised and implemented with the application of proprietary adapters and middleware technologies being predominant among the solutions [17, 18, 19, 20, 21, 22, 23]. Wrapping techniques can be generally classified into the following categories.

Proprietary Adapters

Writing custom software, or glue-code, to unite two disparate applications is by far the oldest form of integration; it does often appear to be a far more attractive solution than redevelopment or reengineering of one or more applications. Over the years, off the shelf software packages serving to bridge technologies have become available; an EJB-COM bridge could, for example, be applied to plug an aging Microsoft-based application into a J2EE implementation. EAI vendors have been selling adapters and adapter toolkits, enabling their platforms to plug into a host of legacy systems, for over a decade. More recently, Sun released the Java Connector Architecture (JCA), a specification aiming to standardize the marketplace of these resource adapters for the java platform; the goal is to mask the complexity behind a layer of indirection akin to JDBC in the world of data management [24]. Though the JCA certainly represents strides in the simplification of this type of solution, the implementation and/or configuration of adapters is a complex, high risk endeavour. Furthermore, this solution will create tightly-coupled, brittle systems with undoubtedly limited life spans; it will only be a matter of time before shifting requirements and technology choices will force the creation of further bridges.

CORBA

The common object request broker architecture is an older middleware technology, which aims to manage much of the underlying plumbing associated distributed object development [25]. As CORBA is language neutral, certain legacy systems can be wrapped with a generic IDL interface and invoked from applications running on entirely different platforms. However, the complexity of projects undertaking this solution is drastically increased, for they involve the introduction of an additional heavyweight, intricate technology. Furthermore, CORBA can only be applied to a relatively small subset of languages and though it does successfully encapsulate complexity behind a published interface, the new platform is mired in a proprietary CORBA framework, thus complicating future modernization efforts.

Messaging Middleware

Messaging middleware technologies, such as the JMS standard or IBM's MQSeries, provide the infrastructure for asynchronous method invocation, rather than direct invocation through an API [26, 27]. The underlying middleware essentially guarantees delivery of a message once it has been sent to the desired destination and allows the client to continue processing without having to wait for a response. This solution is especially attractive for message-oriented legacy systems, for it fosters the creation of a loosely coupled paradigm and enables the construction of complex messaging workflows.

2.1.3.2 System Interface Identification

It can be noted that regardless of the wrapping strategy implemented, the first step must be to analyze the legacy system in order to clearly extract interfaces of the major functional components. In the case of object-oriented languages, such as C++ or object COBOL, this task is trivial; however, it requires additional effort when dealing with older procedural technologies that may be closely intertwined with their environment, such as batch or transaction processing programs running on a mainframe. In certain scenarios, acquiring the understanding of a system interface may require the application of the well known reengineering techniques cited above; however, this case certainly does not reach the complexity of complete white-box transformation. In both [17] and [18], the authors propose tooling to maximize the automation of this process. Once specific legacy components have been identified, their behaviour can be easily specified in terms of well-defined object-oriented interfaces. It can be noted that for the purposes of this paper, it is assumed that the desired interface(s) of a legacy system have been identified and extracted.

2.2 Web Services

2.2.1 Overview

Web services have effectively marked the beginning of the next phase of the internet, where OOP principles are extended in such a way that applications will begin to be composed of reusable components distributed all over the world. The web is being pushed to the next evolutionary step; from providing services to users to providing services to software applications. As depicted below in Figure 2.1, this model is achieved with a classic service-oriented architecture, the three main elements of which are a provider, a client, and a repository. Upon implementing a service, a developer will package it as a web service and publish a WSDL document, which describes the details of the application, in a global repository somewhere on the internet. The binders used in this scenario are typically based on the Universal Description, Discovery, and Integration (UDDI) standard, which essentially provides a ‘yellow pages’ of services that clients can browse based on categories and/or industry groupings [28]. When a client identifies a potential service, the information found in the WSDL document is used to directly connect to then and invoke it using the SOAP protocol.

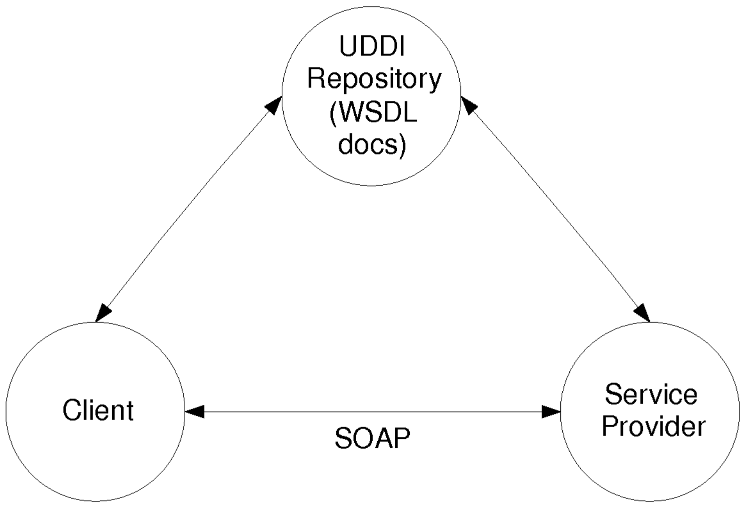


Figure 2. 1 Service-Oriented Architecture

This paradigm can be viewed as a standardization of the distributed computing model, popularized by CORBA, where components are described with IDL interfaces and made available across a network. In this case, however, the use of XML essentially separates the data from the code, thus removing the need for software to fit into proprietary program infrastructures. This data-centricity then naturally leads to the notion of data exchange over the web, for existing, proven, web protocols can be leveraged. As a result, distributed computing is poised to rapidly break free from the shackles of corporate boundaries and explode on a global scale.

It is evident that the invocation of services via SOAP does indeed require a certain amount of underlying plumbing. To this end, a competitive market of web services runtimes, serving to ‘magically’ expose applications as web services, has been created. As depicted in Figure 2.2, given the native interface of a service implementation, a toolkit will automatically generate the components required for communication: a SOAP processing module, dispatcher, (de)serializers, stubs and skeletons, as well as a WSDL document. It can be noted that on the client side, a toolkit compiles this WSDL document to generate the stubs and (de)serializers in the desired language, which may well be different from that of the server implementation. Client code leveraging the service can then be composed and compiled with these generated components.

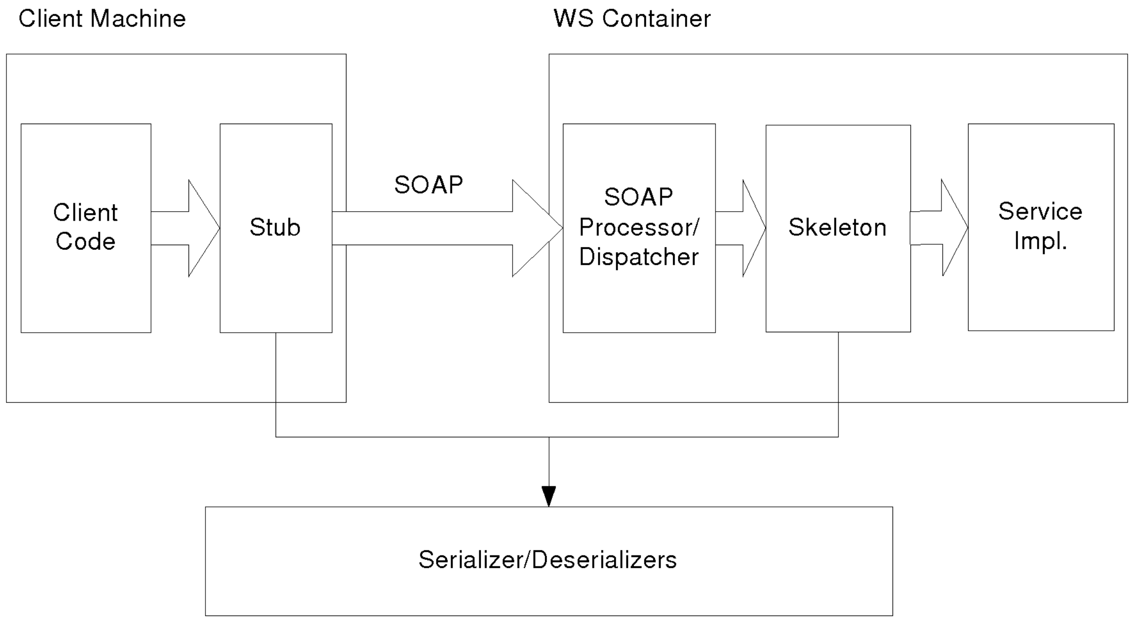


Figure 2. 2 Standard Web Services Deployment

2.2.2 WSDL

An amalgamation of Microsoft's SDL and IBM's NASSL specifications, the web services description language provides a generic means of describing a web service's interface and provides users with a point of contact [29]. As outlined above, the web services model has adopted the paradigm of generating the underlying communication infrastructure based upon application interface description; WSDL can essentially be viewed as an evolution of IDL to an open, interoperable standard. In this case, however, the use of XML over established web protocols extends the communication model far beyond the use proprietary, binary protocols within the intranet. Moreover, unlike the distributed computing technologies of old, the simplicity of the paradigm favours the creation of an open marketplace of free, lightweight toolkits.

In order to maintain the neutrality of the spec, the authors decided to divide it into two logical sections: the abstract and concrete definitions. The basic design principle was to distinctly separate out the description of the data format, transport, and location information from the application level definition, such that it could easily be reused in differing scenarios. The abstract definition essentially describes a service's interface in terms of messages exchanged in a service interaction. Firstly, external type systems (e.g. XML Schema) are referenced to provide data type definitions for the information exchange. These types are then referenced by constructs describing the logical structure of the actual messages to be transmitted over the wire. Finally, these message definitions are then tied together according to a particular interaction model (e.g. request, request/response etc.). In order to successfully invoke a service, a client must also be aware of the expected data format and transport protocol, as well as the physical location of the application; it is bestowed upon on the concrete definition to provide this information. As the specification is designed to be truly extensible, it allows developers to define a set of tags for their desired bindings; it can be noted however, that structures have been provided for popular protocols such as SOAP and HTTP.

2.2.3 SOAP

Initially developed by Microsoft, SOAP is a simple protocol for messaging and remote procedure calls; as it works on existing transports (HTTP, SMTP etc.), it has rapidly become the de-facto standard for XML message format definition in the world of web services [31]. At it core, SOAP has a truly trivial structure with each message merely containing a header and body element. The former contains meta-information describing how the message should be processed, while the latter contains the contents of the message itself. Though the make-up of each of these tags is generally arbitrary, it can be noted that some additional basic structure is defined for the body in the case of RPC communication. In the end, given a WSDL description specifying SOAP as the message format, a toolkit will generate a wrapper for the desired application to produce and consume SOAP messages.

2.2.3.1 SOAP Encoding

In the early stages, the framers of the SOAP spec recognized the need for a means to describe data types in RPC communication. As XML Schema had not yet been completed, an encoding scheme, dubbed SOAP encoding, was devised to serialize object graphs into XML messages. These encoding rules were based on a data model defined to represent application defined structures as a directed, edge-labelled graph of nodes. These rules addressed both object references and arrays, two facets which were eventually omitted from the ratified XSD specification. As a result, the WSDL spec leverages both XSD and SOAP encoding for programmatic type serialization, which, as we will discuss in the ensuing chapter, creates an array of interoperability concerns.

2.2.3.2 Binary Data

It is evident that packing binary data into a SOAP envelope is indeed infeasible, for bytes with special meanings (e.g. '<') will eventually appear. Moreover, encoding the data as Base64 is not an ideal, for message size is expanded by a third, thus degrading performance. Since the successful transmission of binary data is indeed of great importance to the future success of web services, it has been addressed in two specifications: SOAP with Attachments (SwA) and WS-Attachments [33, 34]. The former outlines how a SOAP message can be carried within a multipart MIME message in such a way that the SOAP processing rules are preserved. In this case, the binary data is appended at the end of message and referenced from the SOAP body with URIs. There are, however, two main drawbacks to this model. Firstly, MIME can be heavyweight and is therefore inappropriate for small and embedded devices. More importantly, SwA cannot handle data streaming; for any sort of multi-media application, it is clear that the receiver should not have to buffer the entire attachment before processing it. As such, though the WS-Attachments spec follows the model of URI referencing, it mandates the use of DIME, a binary message format [35]. As attachments can be broken up across multiple DIME 'chunks' of variable length, this proposal is well suited to dynamically generated and/or streaming content.

2.3 Server Side Computing

Over the last two decades, enterprise-class installations have evolved from large, unmanageable monoliths to multi-tiered, distributed, component architectures; a classic three-tier deployment is pictured below in Figure 2.3. It can be noted that in order to offer both high availability and scalability, each of the three layers is typically distributed across a number of machines, thus introducing a great deal of complexity into the system. As we will outline below, the application server was therefore born from the need to handle much of this essential underlying plumbing.

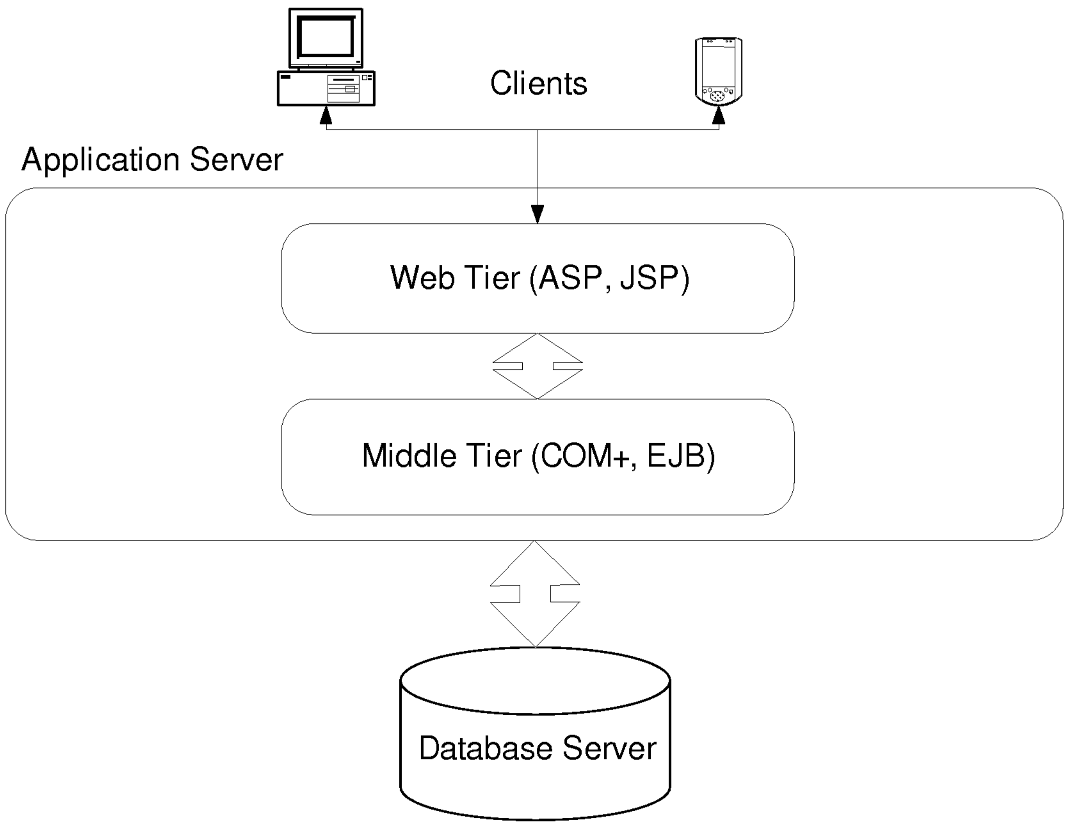


Figure 2. 3 Traditional 3-tier System Architecture

2.3.1 Web Tier

A web container is typically used both to serve up static content and process incoming client requests. Technologies such java servlets, JSP, and ASP are used to extend the web server in such a way that requests are first inspected and delegated to the appropriate middleware component; a presentation page is then selected and rendered before being returned to the user.

2.3.2 Middle Tier

As outlined above, middleware is the technology that facilitates the integration of components in a distributed system. It is traditionally defined as the software that allows elements of applications to interoperate across network links, despite differences in underlying protocols, system architectures, operating systems, databases, and other application services. The framework required for the implementation and maintenance of a secure, transactional, scalable, highly available enterprise level deployment is indeed incredibly complex. As this high-end middleware requires a tremendous amount of expert knowledge, companies have moved away from building their own, choosing to focus instead on their core competency. Technologies such as EJB or COM+ are therefore leveraged to provide a seemingly endless list of services: from support for resource pooling, networking, and caching to load-balancing and fail-over to application level issues such as security and transactions [36]. Developers are essentially left to focus on the business logic, which is then automatically enterprise-enabled with a seamless deployment into the application server.

Chapter 3
Introducing Interoperability into the Web Services Model

3.1 Introduction

As alluded to previously, the additional veneer provided by XML web services is expected to unite a completely heterogeneous computing world. In order to accomplish this elusive task, it is evident that a huge amount of XML infrastructure, originating from a host of different toolkits supporting different technologies, must be generated. This new paradigm is indeed an outstanding one with vast potential, however, it will only be widely adopted if interoperability is simple to achieve; code generated at the client must easily produce SOAP messages that will be fully understood at the server. However, coercing any of the sixty plus web services toolkits currently available into working together to perform even the simplest of operations has proven to be a truly daunting task.

As the WSDL specification is responsible for the formalization of the necessary underlying plumbing, there is little doubt that it is the root cause of the current interoperability problems. The spec is so excessively verbose and complex that not only do web service programmers have trouble grasping the meaning of the options available to them, but toolkit implementers themselves have differing interpretations and are thus struggling to provide a standard set of features. As with any new technology, there will be an extended period of testing and improvement before stability is finally reached. However, the major concern in this case is that WSDL is fundamentally flawed at the conceptual level; its evolution is undoubtedly on the wrong path. We strongly believe that true interoperability will only be achieved via message level schema validation. The adoption of this philosophy would provide endpoints with a precise blueprint for network communication, thus automatically removing any ambiguity. It is with this concise objective in mind that this chapter details the major interoperability pitfalls of the current WSDL spec and outlines several crucial amendments. Moreover, it can be noted that the W3C is notorious for contorting perfectly legible specs into bloated, complex documents, the entirety of which is only understood by a select few. If current trends continue, it is evident that only the software giants, those driving the W3C, will have the knowledge and resources to compete in the world of web services' plumbing. The proposed simplifications will therefore also serve to rescue the dream of an open marketplace of compatible toolkits from the cruel boot of capitalism.

3.2 WSDL 1.1 Abstract Definition

3.2.1 Background

As mentioned in the previous chapter, WSDL 1.1 is separated into two main portions: the abstract definition and concrete bindings. The ensuing portions of this section will outline and provide solutions to two major interoperability flaws of the current abstract definition. As a frame of reference, a complete abstract service description, containing <wsdl:types>, <wsdl:message>, and <wsdl:portType> constructs, is provided below.

```
<definitions targetNamespace="http://www.example.com/PO/wsdl"
             xmlns:poxsd="http://www.example.com/PO/xsd"
             xmlns:tns="http://www.example.com/PO/wsdl"
             xmlns="http://schemas.xmlsoap.org/wsdl">

    <types>
        <schema targetNamespace="http://www.example.com/PO/xsd"
                xmlns="http://www.w3c.org/2001/XMLSchema">

            <element name="PurchaseOrder">
                <complexType>
                    <sequence>
                        <element name="buyer" type="string" />
                        <element name="item" type="string" />
                        <element name="date" type="date" />
                    </sequence>
                 </complexType>
            </element>

            <complexType name="Person">
                <sequence>
                    <element name="name" type="string" />
                    <element name="department" type="string" />
                    <element name="phoneExtension" type="int" />
                </sequence>
            </complexType>
        </schema>
    </types>

    <message name="ProcessPORequest">
        <part name="Param1" element="poxsd:PurchaseOrder" />
        <part name="Param2" type="poxsd:Person" />
    </message>
    <portType name="ProcessPOPortType">
        <operation name="ProcessPO">
            <input message="tns:ProcessPORequest" />
        </operation>
    </portType>
    .
    .
    .
</definitions>
```

Figure 3.1 Abstract definition of 'PurchaseOrder' endpoint

This example describes a service named 'ProcessPO', which accepts two arguments, 'PurchaseOrder' and 'Person', and does not return a response. It is assumed in this case that the business rules governing the endpoint require that information regarding the individual who created the purchase order be submitted as a separate parameter. One can envision this type of service having literally thousands of applications across a multitude of industries. For example, a project manager on a construction site could use a wireless device to submit purchase orders to her suppliers while simultaneously sending a copy to her company's home office to update their records in real time. It can be noted that the <wsdl:types> contains the schema definitions for the content of the service. As we will soon discuss in more detail, the specification claims these definitions are either abstract or concrete; in the former case, a set of encoding rules must be applied to them in order to obtain a specific XML format while in the latter case, the XML instance must conform to the schema.

3.2.2 Interoperability Problems

3.2.2.1 Type System Ambiguity

Since one of the initial objectives of the web services' paradigm was to leverage existing web protocols in order to provide a generic XML-based RPC mechanism, the decision was made that the intermediate messages should be strongly typed. As a result, the <wsdl:types> construct depicted above in Figure 3.1 serves as a container for the abstract parameter data type definitions referred to in the <wsdl:message>. The current specification claims that "for maximum interoperability and platform neutrality, WSDL prefers the use of XSD as the canonical type system, and treats it as the intrinsic type system" [29]. Despite its "preference" for XML Schema, the spec's set of guidelines, outlined in section 2.2, for encoding abstract types using XSD is alarmingly vague and incomplete. For toolkits to interoperate successfully, they must have a complete set of standardized rules in order to create consistent XML Schema descriptions from each programmatic type system. We will delve into the XSD mapping issue in greater detail below when WSDL binding encoding mechanisms are scrutinized.

It has been noted that WSDL prefers XML Schema; however, in the spirit of extensibility, it was decided that other type systems could also be used “since it is unreasonable to expect a single type system grammar to describe all abstract types both present and future” [29]. It is true that a number of standards organizations have published credible type system specifications; in addition to the W3C’s XML Schema spec, ISO released Relax NG schema, and OASIS issued TREX schema [37,38]. However, it is absolutely ludicrous to believe that WSDL, the core spec in web services’ plumbing, is completely uncommitted to a type system. To ensure an open, interoperable world of web services, this decision is essentially forcing all toolkit vendors to successfully support multiple type systems; a feat that if not impossible, is completely unreasonable. Apart from compounding the mapping problem outlined above, complicating toolkit implementation to allow for numerous type systems that all essentially provide the same functionality is utterly redundant.

3.2.2.2 The Message Construct: Duplicate Representation

It can be observed in Figure 3.1 that once a type system has been selected, the <wsdl:message> construct serves as a placeholder for the required content in a service invocation. Each information item is represented within the <wsdl:message> using a peculiar element known as a <wsdl:part>, which represents a logical abstraction of the content of the message. It is not until the bindings are inspected that it is known precisely what the <wsdl:part> represents. For example, a <wsdl:part> could, using any type system, portray an RPC parameter, an XML document, or even some form of binary information (image, audio, video etc.). As a result, depending on the bindings, the ‘ProcessPORequest’ <wsdl:message> outlined above represents a service that accepts either two xml documents or two RPC parameters for processing. In providing this unified mechanism for a first-class description of both RPC and document style operations, the <wsdl:message> essentially serves as a simple yet inexpressive type system for content representation. However, several capabilities are completely absent; in addition to the inability to be shared between <wsdl:message>s, <wsdl:part>s cannot be defined as optional or have an associated multiplicity. These limitations have led to discussion to extend the functionality of the <wsdl:message>; however, the mandate of WSDL is to provide a succinct set of operational types, thus calling into question why any time at all has been spent developing a representational one. As discussed in section 4.2.3.2 below, the logical choice in this case is to reap the rewards of the excellent work done by the XML Schema working group.

Expressiveness aside, it can be shown that the <wsdl:part>s interact with the chosen type system in an unnecessarily flexible manner. Each <wsdl:part> refers to a type with a specific type referencing attribute; as the authors have a slight predilection to XML Schema, they have provided two attributes with which to refer to XSD. It can be observed from the example outlined above that the 'element' attribute refers to a schema element while the 'type' attribute refers to a globally defined schema complexType. It can be noted that in order to accommodate other type systems, the set of typing referencing attributes is extensible under the condition that they are in a namespace other than that of WSDL. The specification outlines that the 'type' attribute serves as an alternative syntax when the message contents are sufficiently complex; however, in reality both forms are equally expressive. As the current development paradigm dictates that developers should rarely, if ever, interact with a WSDL document, one can question the validity of providing two semantically equivalent type referencing mechanisms. The option simply serves to unnecessarily add to the implementation complexity of web services' toolkits. Furthermore, it will be later shown that <wsdl:part>s interact with bindings in a non- trivial manner and allowing two type referencing mechanisms only serves to drastically complicate the bindings.

Finally, it is often the case that <wsdl:message>s are superfluous and appear only for syntactic reasons. The following excerpt from section 1.2 of the WSDL 1.1 specification exemplifies the commonly used 'single part' authoring style.

```
<definitions targetNamespace="http://example.com/stock/wsdl"
             xmlns:tns="http://example.com/stock/wsdl"
             xmlns:stxsd="http://example.com/stockquote.xsd"
             xmlns="http://schemas.xmlsoap.org/wsdl">

   <types>
       <schema targetNamespace="http://example.com/stockquote.xsd"
```

```
                xmlns="http://www.w3.org/2000/10/XMLSchema">

            <element name="TradePriceRequest">
                <complexType>
                    <all>
                        <element name="tickerSymbol" type="string"/>
                    </all>
                </complexType>
            </element>

            <element name="TradePrice">
                <complexType>
                    <all>
                        <element name="price" type="float"/>
                    </all>
                </complexType>
            </element>
           </schema>
    </types>

    <message name="GetLastTradePriceInput">
           <part name="body" element="stxsd:TradePriceRequest"/>
    </message>

    <message name="GetLastTradePriceOutput">
           <part name="body" element="stxsd:TradePrice"/>
    </message>

    <portType name="StockQuotePortType">
           <operation name="GetLastTradePrice">
                <input message="tns:GetLastTradePriceInput">
                <output message="tns:GetLastTradePriceOutput"/>
           </operation>
    </portType>
    .
    .
    .
</definitions>
```

Figure 3.2 Abstract definition portraying 'single part' authoring style

The fact that the names of the <wsdl:message>s closely mirror the names of the schema elements provides a strong indication that they could be dropped. It is evident that no loss of information would occur if the operation simply referred directly to the XSD constructs. Furthermore, the <wsdl:part>s are commonly named 'body', the generality of which provides a hint of the artificial nature of the construct.

As it stands now, the <wsdl:message> is an unintuitive, redundant construct, the presence of which not only serves to further complicate an already bloated specification, but precludes the possibility of validating SOAP messages in their entirety. As a result, the following two sections will outline in detail a proposal for the removal of the <wsdl:message> construct in favour of solely using XML Schema.

3.2.3 Interoperability Solutions

3.2.3.1 Mandating XML Schema

As mentioned previously, by providing each vendor with their choice of typing mechanisms, the WSDL authors have strayed from the path toward a world of successfully interoperating toolkits. To attain this goal, we believe it is imperative that WSDL mandate XSD as the sole present form of data representation. Despite the controversy surrounding its complexity, XML Schema was ratified as an exceptionally expressive type system that has rapidly become the de-facto industry standard. In fact, in the last two years, it has been the only typing mechanism integrated into virtually all relevant XML technologies. As it is, however, reasonable to envision that another typing mechanism will one day usurp XML Schema as the industry standard, the WSDL specification must be accordingly extensible; an issue that is dealt with in detail below.

3.2.3.2 Message Construct Removal

Underpowered and sandwiched between more interesting and semantically rich constructs (<wsdl:operation> and <wsdl:type>), the inexpressive <wsdl:message> element has been shown to exist for mainly syntactic reasons. Having proposed that WSDL adopt XSD as its type system of choice, the next logical step is to remove the <wsdl:message> construct entirely in favour of XML Schema, thus providing a host of compelling advantages. The ensuing document incorporates these proposed changes to describe the endpoint depicted in Figure 3.1.

```
<definitions targetNamespace="http://www.example.com/PO/wsdl"
              xmlns:poxsd="http://www.example.com/PO/xsd"
              xmlns="http://schemas.xmlsoap.org/wsdl">

   <types>
      <schema targetNamespace="http://www.example.com/PO/xsd"
                 xmlns:tns="http://www.example.com/PO/xsd"
                 xmlns="http://www.w3c.org/2001/XMLSchema">

         <element name="ProcessPORequest">
            <complexType>
```

```
                    <sequence>
                        <element name=”PO” type=”tns:PurchaseOrder”>
                        <element name=”Person” type=”tns:Person”>
                    </sequence>
                </complexType>
            </element>

            <complexType name=”PurchaseOrder”>
                <sequence>
                    <element name=”buyer” type=”string” />
                    <element name=”item” type=”string” />
                    <element name=”date” type=”date” />
                </sequence>
            </complexType>

            <complexType name=”Person”>
                <sequence>
                    <element name=”name” type=”string” />
                    <element name=”department” type=”string” />
                    <element name=”phoneExtension” type=”int”>
                </sequence>
            </complexType>
        </schema>
    </types>

    <portType name=”ProcessPOPortType”>
        <operation name=”ProcessPO”>
            <input element=”poxsd:ProcessPORequest” />
        </operation>
    </portType>
    .
    .
    .
</definitions>
```

Figure 3.3 Abstract definition of ‘PurchaseOrder’ endpoint with XSD-only syntax

It can be observed that the <wsdl:message> construct has been replaced with an <xsd:element> carrying the operation name with ‘Request’ appended to it. The <wsdl:part>s have, in turn, been supplanted with a sequence of child <xsd:element>s, each of which must either refer to a schema simple type or a schema compound type defined in any namespace. In order to refer to the newly defined ‘ProcessPORequest’ element from within the <wsdl:portType>, we simply propose that the ‘message’ attribute found on the <wsdl:input>, <wsdl:output>, and <wsdl:fault> constructs be aptly renamed to ‘element’. As we will outline below, this proposal represents a clean, simplistic solution to all the issues outlined in section 3.2.2.2.

The existing <wsdl:message> tag has two primary design objectives: to simultaneously describe both RPC and document style operations and to act as a neutral container for types from any available typing mechanism. In order to provide programming model flexibility, we understood the importance of seamlessly propagating the former issue into our proposal. As a result, it can be noted in Figure 3.3 that the 'Person' and 'PurchaseOrder' parameters could indeed either represent programmatic objects destined for serialization or XML documents to be processed directly. In fact, the inclusion of the schema element wrapper for the operation yields a far more descriptive, accurate view of the service and introduces the possibility of attaining the goal of message level schema validation.

With respect to the second design criteria, since the future success of web services depends on the acceptance of XML Schema as the sole representational type system, we firmly believe that it is no longer required. However, in the case that XSD is one day replaced as the industry standard, our proposal must allow WSDL to be extensible. This issue is accounted for in a similar manner to the current specification by moving the extensible set of type referencing attributes from the defunct <wsdl:part> tag to the <wsdl:input>, <wsdl:output>, and <wsdl:fault> elements within an <wsdl:operation>. Any compelling future type system will provide structuring constructs at least as powerful as the <wsdl:message>, thus its removal won't be regarded as a limitation; on the contrary, the ability to use the full power of the native type system will be energizing, as is the case for XML Schema.

Furthermore, the use of the <wsdl:message> to describe a service's structure is merely recreating a far less expressive version of XML Schema. The simplification of the specification to use schema elements in place of <wsdl:part>s implicitly gains a rich set of features; the 'minOccurs' and 'maxOccurs' attributes can be used to specify cardinality constraints and the 'ref' attribute allows for the sharing of <xsd:element>s at any level of the description. Moreover, schema has a host of more advanced features such as type inheritance, substitution groups, and support for extensible content models, all of which could well be exploited in the future. It can also be noted that the elimination of the 'type' and 'element' attributes as type-referencing mechanisms in favour of simply referring to XSD 'complexType' drastically simplifies the lives of toolkit developers.

In addition to radically improving the abstract definition, it can be noted that this proposal also serves to simplify the bindings on two levels. Firstly, since binding authors need to take into account all aspects of an operation before they can fully specify how to bind it to a specific protocol, their work instantly becomes a great deal more difficult when different constructs (<wsdl:message>, <xsd:element> / <xsd:complexType>) or type systems are used as part of the same operation definition. The simplicity and syntactic clarity of the XML Schema representation clearly makes this task far less painful. Secondly, there are certain scenarios in the WSDL 1.1 bindings (e.g. <soap:header>) where special syntax is introduced to select specific <wsdl:part>s. Adopting this schema-centric proposal would mean foregoing this syntax in favour of proven, well-known XML technologies such as XPath.

Finally, a core issue that is poorly taken up in the current spec is the representation of binary data. As one could envision literally countless applications involving the transfer of MIME types, we have devised an efficient, elegant solution to address the problem. As seen in the example below, the xsd:hexbinary simple type is currently used to represent binary data in the abstract definition, thus making it impossible to determine what type of data is being sent (image, video, audio etc.) until the bindings are inspected. With respect to the bindings themselves, chapter two outlined the major reasons why it is unfeasible to efficiently embed binary data into a SOAP message. As a result, the accepted standard is to use the notion of "attachments", where information is appended after the SOAP envelope and referenced with URIs. There are a number of possible data formats to which this methodology could be mapped and WSDL 1.1 mandates the use of multipart MIME messages. In describing a service that accepts both a patient record and an x-ray image, the ensuing document exemplifies how binary information is treated in the current specification.

```
<definitions targetNamespace="http://www.example.com/patient/wsdl"
              xmlns:tns="http://www.example.com/patient/wsdl"
              xmlns:paxsd="http://www.example.com/patient/xsd"
              xmlns:xsd="http://www.w3c.org/2001/XMLSchema"
              xmlns="http://schemas.xmlsoap.org/wsdl">

   <types>
       <schema targetNamespace="http://www.example.com/patient/xsd"
                xmlns="http://www.w3c.org/2001/XMLSchema">

           <complexType name="Patient">
               <sequence>
                   <element name="name" type="string" />
```

```
                        <element name="age" type="int" />
                        <element name="date" type="date" />
                    </sequence>
                </complexType>
            </schema>

        </types>

        <message name="ProcessPatientRequest">
            <part name="param" type="paxsd:Patient"/>
            <part name="param1" type="xsd:hexBinary">
        </message>

        <portType name="ProcessPatientPortType">
            <operation name="ProcessPatient">
                <input message="tns:ProcessPatientRequest" />
            </operation>
        </portType>

        <binding name="PatientBinding" type="tns:ProcessPatientPortType">
            <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
            <operation name="ProcessPatient">
                <soap:operation soapAction="http://www.example.com/patient" />
                <input>
                    <mime:multipartRelated>
                        <mime:part>
                            <soap:body parts="param">
                        </mime:part>
                        <mime:part>
                            <mime:content part="param1" type="image/jpeg/">
                        </mime:part>
                    </mime:multipartRelated>
                </input>
            </operation>
        </binding>
</definitions>
```

Figure 3.4 Description of binary data using WSDL 1.1 constructs

At service discovery time, potential clients search and/or examine the abstract definitions of WSDL files in order to find an endpoint that satisfies their requirements. The specification that a parameter is of type xsd:hexbinary does not yield any pertinent information; it could be an image of the patient's chart, an audio recording of the doctor's diagnosis, or even a video recording of an actual procedure. As observed in the preceding example, by including this valuable portion of the service description in the <mime:content> tag, the binding has gone far beyond its assigned task of providing transport and data format information, thus breaching a fundamental design requirement of the specification. Some of the original WSDL 1.1 authors claim that this issue can be resolved by recognizing MIME as a distinct type system deserving of its own message typing attribute within the <wsdl:part>. For example:

```
<message name="ProcessPatientRequest">
    <part name="param" type="paxsd:Patient"/>
    <part name="param1" mime:type="image/jpeg">
</message>
```

Figure 3.5 Description of binary data using amended <wsdl:message> construct

This scenario is the only compelling argument in favour of retaining the message construct; MIME types are indeed prevalent and could be considered a specialized, distinct type system. However, we strongly believe that the overall expressive power and simplicity obtained from removing the <wsdl:message> and encapsulating binary information within XML Schema far outweighs any gains in syntactic clarity. It is assumed that any future replacements for XSD will be equally capable of incorporating raw data.

The previous chapter provided a brief overview of the WS-Attachments spec, which essentially specifies how to package both the SOAP envelope and binary data into a DIME message. Microsoft's Mike Deem went on to author a draft detailing exactly how to incorporate these concepts into WSDL [39]. He crafted new schema complexType definitions, defined in their own namespace, to incorporate both separate XML documents and MIME types. Based upon this work, we have devised a proposal, exemplified below by a modified version of Figure 3.4, for the incorporation of binary data into WSDL.

```
<definitions targetNamespace="http://www.example.com/patient/wsdl"
              xmlns:tns="http://www.example.com/patient/wsdl"
              xmlns:paxsd="http://www.example.com/patient/xsd"
              xmlns:xsd="http://www.w3c.org/2001/XMLSchema"
              xmlns="http://schemas.xmlsoap.org/wsdl">

    <types>
        <schema targetNamespace="http://www.example.com/patient/xsd"
                xmlns:content="http://schemas.xmlsoap.org/ws/2002/04/content-type/"
                xmlns:dime="http://schemas.xmlsoap.org/ws/2002/04/dime/wsdl/"
                xmlns="http://www.w3c.org/2001/XMLSchema">

            <import namespace="http://schemas.xmlsoap.org/ws/2002/04/content-type/"/>

            <element name="ProcessPatientRequest">
                <complexType>
                    <sequence>
                        <element name="Patient" type="paxsd:Patient">
                        <element name="XRay" type="paxsd:XRay">
                    </sequence>
                </complexType>
            </element>
            <complexType name=”Patient”>
                    <sequence>
                        <element name=”name” type=”string” />
                        <element name=”age” type=”int” />
                        <element name=”date” type=”date” />
                    </sequence>
            </complexType>

            <complexType name=”ReferencedBinary”>
                <simpleContent>
                    <extension base=”hexBinary”>
                        <attribute name=”location” type=”anyURI” use=”optional”>
                    </extension>
                <simpleContent>
            </complexType>

            <complexType name=”XRay”>
                <simpleContent>
                    <restriction base=”paxsd:ReferencedBinary”>
                        <annotation>
                            <appinfo>
                                <content:mediaType type=”image/jpeg” />
                            </appinfo>
                        </annotation>
                    </restriction>
                </simpleContent>
            </complexType>

        </schema>

    </types>

    <portType name=”ProcessPatientPortType”>
        <operation name=”ProcessPatient”>
            <input element=”poxsd:ProcessPatient” />
        </operation>
    </portType>

    <binding name=”PatientBinding” type=”tns:ProcessPatientPortType”>
        <soap:binding style=”rpc” transport=”http://schemas.xmlsoap.org/soap/http” />
        <operation name=”ProcessPatient”>
            <soap:operation soapAction=”http://www.example.com/patient” />
            <input>
                <dime:message layout=”http://schemas.xmlsoaop.org/ws/2002/04/dime/closed-layout”
                wsdl:required= ”true”>
                <soap:body elements=”Patient” use=”Literal”>
            </input>
        </operation>
    </binding>
    .
    .
    .
</definitions>
```

Figure 3.6 Description of binary data using proposed constructs

In this example, it is immediately evident that the service is expecting a jpeg image of an x-ray for it is concisely defined in the abstract definition. This added description is obtained by extending the xsd:hexBinary base type in two ways to create a new XRay complexType. First, as the standard method of incorporating external XML documents and binary data into a SOAP message is via attachments, a mechanism must be defined by which the envelope can reference the appended information. This objective is accomplished by extending xsd:hexBinary to include a 'location' attribute of type xsd:anyURI. Furthermore, a 'pseudo-facet' has been defined which, similar to all facets defined in XML Schema (length, pattern, totalDigits etc.), restricts the content of the data. As is required by XSD, these facet elements will always appear within an xsd:appinfo tag who's parent element, xsd:annotation, may appear in an xsd:restriction construct; in this particular case, the restriction is then applied to the xsd:hexBinary simple type. The presence of this <content:mediaType> pseudo-facet signifies that the value space of the parent element is constrained to known MIME types as defined in RFC 2616. In Figure 3.6 above, it is the facet's 'value' attribute indicates that an 'image/jpeg' is to be passed to the service.

As more detailed typing information is now included in the abstract definition, it is possible to collapse the current verbose MIME bindings into a single element. This simplification is based upon the assumption that any data types containing pseudo-facets will be included as attachments while the "standard" information is encapsulated within the SOAP envelope. Depicted above in Figure 3.6, the proposed modifications manifest themselves into the single <dime:message> tag. An extensible 'layout' attribute has been specified, which as the name implies, allows for specific message format configuration; for example, an open content layout allows for the inclusion of attachments which aren't referenced from the SOAP envelope while a closed content layout does not. In the case that a MIME format is preferred, the <dime:message> tag would be replaced with a similar <mime:message> construct complete with 'layout' and 'wsdl:required' attributes. It can be noted that it would be trivial to incorporate future binary data formats into WSDL simply by the creation of new descriptive elements.

3.3 WSDL 1.1 Bindings

3.3.1 Background

3.3.1.1 Definition

Having proposed improvements to the WSDL 1.1 abstract definition constructs, the bindings will now be scrutinized. As mentioned in the second chapter, the <wsdl:binding> tags are responsible for specifying the concrete data format and transport protocol for each <wsdl:portType>. In this section, we will focus on the SOAP binding, for it represents the de-facto xml protocol of the future and is far more expressive than either of the HTTP or MIME equivalents. We will begin by describing the salient features of the binding, outlined below, for the abstract definition provided in Figure 3.1.

```
<definitions targetNamespace="http://www.example.com/PO/wsdl"
              xmlns:poxsd="http://www.example.com/PO/xsd"
              xmlns:tns="http://www.example.com/PO/wsdl"
              xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
              xmlns="http://schemas.xmlsoap.org/wsdl">
   .
   .
   .
   <binding name="ProcessPOBinding" type="ProcessPOPortType">
      <soap:binding style="RPC" transport="http://schemas.xmlsoap.org/soap/http" />
      <operation name="ProcessPO">
         <soap:operation soapAction="http://www.example.com/ProcessPO" />
         <input>
            <soap:body use="encoded" encodingStyle=
               "http://www.w3.org/2002/12/soap-encoding">
         </input>
      </operation>
   </binding>
   .
   .
   .
</definitions>
```

Figure 3.7 WSDL SOAP binding for 'PurchaseOrder' endpoint

It can be observed that the <wsdl:binding> element references a specific <wsdl:portType>; in the case that there is more than one <wsdl:binding> per <wsdl:portType>, they are considered alternatives. We will now provide an overview of the more interesting extension elements comprised in the SOAP binding.

Firstly, the <soap:binding> specifies the style of the message interaction, either ‘RPC’ or ‘document’. As the name implies, the RPC style signifies that the service is a function-oriented endpoint; the SOAP messages essentially describe remote procedure calls to be invoked on the remote machine. As a result, the preceding example signifies that the ‘PurchaseOrder’ and ‘Person’ constructs specified in the abstract definition in Figure 3.1 are to be serialized into native objects before being passed to the ‘ProcessPO’ method. As there is no <wsdl:output> specified, the method could have the following java signature:

```
void ProcessPO(PurchaseOrder i_objPurchaseOrder, Person i_objPerson);
```

In the case that the binding specified a document style, the service would then represent a message-oriented endpoint; the SOAP messages simply contain XML document(s) for processing. If the preceding example had been a document endpoint, the ‘PurchaseOrder’ and ‘Person’ definitions would have represented distinct XML documents to be manipulated directly with the DOM (or SAX) API:

```
void ProcessPO(Document i_objPurchaseOrderDoc, Document i_objPersonDoc);
```

It can be noted that in addition to providing the style, the <soap:binding> also defines the underlying message transport protocol; a task accomplished by specifying a well known URI with the ‘transport’ attribute. As it is currently the core application protocol of the world wide web, the vast majority of toolkits predominantly support only HTTP; however, it is inevitable that implementations supporting other technologies, such as SMTP, FTP, TCP etc., will slowly begin to emerge.

According to the WSDL 1.1 specification, the <soap:body> element specifies precisely how the <wsdl:part>s are to appear inside the body of a SOAP message. The most compelling feature of this element is the ‘use’ attribute, which can carry either of two values: ‘encoded’ or ‘literal’. As mentioned previously, the <wsdl:types> tag contains either abstract type definitions or concrete schema definitions. In the former case, encoded bindings must be used in order to serialize the types to XML with a set of known encoding rules specified by the ‘encodingStyle’ attribute. The most widespread encoding scheme implemented in existing toolkits is SOAP encoding, which addresses XML Schema's two main weaknesses: implicit support for both arrays and object references. It can be noted that given a set of abstract types, a number of encoding mechanisms, including SOAP encoding, allow variations in the resulting message format. In this paradigm, known as 'reader makes right', it is up to the reader of the message to understand all the possible variations. In order to avoid supporting the differing message possibilities, the <wsdl:types> must contain concrete type definitions. In this case, known as 'writer makes right', literal bindings are specified and the parameters in the SOAP message must conform exactly to their specified schema definitions.

3.3.1.2 SOAP Message Creation

Having completed an overview of the WSDL 1.1 SOAP binding, the guidelines used to create meaningful SOAP messages will now be discussed. Given the differing binding options, the specification outlines a set of rules in order to provide toolkit implementers with knowledge of the wide variety of message formats they must support.

Firstly, if the operation style is specified to be RPC, <wsdl:part>s are grouped in the same order as the parameters of the call within a wrapper element carrying the name of the operation. Furthermore, each <wsdl:part> is also wrapped with an accessor element named identically to the corresponding parameter of the call. In the case that a document operation style is specified, there are no additional wrappers and the <wsdl:part>s simply appear directly under the SOAP body element.

When an encoded binding is specified, it is interesting to note that the spec claims that each <wsdl:part> must reference an abstract type using the 'type' attribute. Upon completion of the encoding algorithm, the concrete type definition is appropriately inserted into the message. In the case of literal bindings, each part references a concrete schema definition using either one of the 'element' or 'type' attribute. In the former case, the referenced element will appear directly under the SOAP Body element in document bindings or under an accessor element named after the <wsdl:part> in RPC style. In the case that an XML Schema type is referenced, it becomes the schema type of the enclosing element; the SOAP Body in document style or part accessor in RPC style. It can be noted that the value of the encodingStyle attribute may be used in literal bindings to indicate that the concrete schema format was derived using a particular encoding. If this attribute is specified, however, only the specified variation is supported.

In order to better illustrate these guidelines, we have outlined SOAP sample messages invoking the service outlined in Figure 3.1 for the two most prevalent binding scenarios: RPC/ENC and DOC/LIT. The RPC encoded bindings are meant for use in the case of remote procedure calls while the document literal bindings are meant for endpoints expecting to process the XML documents themselves.

```
<env:Envelope xmlns:env="http://www.w3.org/2002/12/soap-envelope">
    <env:Body>
        <ProcessPO>
            <param2>
                <po:Person xmlns:po="http://www.example.com/PO/xsd">
                    <name>Craig</name>
                    <department>Purchasing</department>
                    <extension>512</extension>
                <po:/Person>
            </param2>
        <ProcessPO>
    </env:Body>
</env:Envelope>
```

Figure 3.8 SOAP message generated from RPC/encoded binding

```
<env:Envelope xmlns:env="http://www.w3.org/2002/12/soap-envelope">
    <env:Body xmlns:po="http://www.example.com/PO/xsd">
        <po:name>Craig</po:name>
        <po:department>Purchasing</po:department>
        <po:extension>01/01/2003</po:extension>
    </env:Body>
</env:Envelope>
```

Figure 3.9 SOAP message generated from document/literal binding

In the first case, it can be noted that the 'PurchaseOrder' type has not been included in the message; the astute reader will realize that this omission is due to its <wsdl:part> utilizing the 'element' type-referencing attribute. Another main point of interest in the RPC case is the two added layers of element wrapping. Finally, it can be noted that since the operation does not include any object references or arrays, the SOAP encoding algorithm has no affect on the type definition (ie the <po:Person> element matches the schema definition). When using literal bindings, as depicted in Figure 3.9, any <wsdl:part> using the 'type' attribute must be the sole parameter in the invocation; in this case, the <env:Body> element takes on the

<po:Person> type. As a result, the example above is included for illustration purposes only as the vast majority of toolkits would have thrown an error upon compilation of the WSDL document in this case. However, had the Person type been referenced using the 'element' attribute, both constructs could have been successfully included.

It can be noted that we felt this rather in depth overview was required in order to accurately portray the excessive complexity of the specification. It is no wonder that interoperability is a problem when toolkits are faced with the task of applying this convoluted set of rules in order to generate SOAP messages. Our main objective is to abolish these rules in favor of simply generating a schema definition to which the SOAP messages required for successful invocation must conform; the SOAP binding should, in essence, no longer be responsible for detailing data format. We will now build upon the removal of the <wsdl:message> construct and explore ways to improve the binding definition in order to achieve this task.

3.3.2 Interoperability Problems

3.3.2.1 Binding Dependence upon Abstract Definition

One of the fundamental design decisions made in the composition of the WSDL 1.1 specification was the clear definition of completely detached abstract definitions and concrete bindings. The idea was that, given the application of the underlying protocol made sense, virtually any binding could be applied to any abstract definition. It is evident that the specification violates this concept for not only do encoded bindings forbid the use of the <wsdl:part> 'element' type-referencing attribute, but the actual concrete message format itself is also dependent on the selection of this attribute. Furthermore, Figure 3.9 demonstrated that even the validity of certain concrete formats is dependent upon the contents of the <wsdl:part>. The spec clearly contradicts itself without any valid justification for its unmistakable breach of contract. Assuming toolkit vendors decipher the spec correctly, they must then support these unintuitive dependencies, thus increasing the probability of encountering interoperability problems. It can be noted that our proposal to remove the <wsdl:message> construct partially rectifies this problem by ensuring that <wsdl:type> definitions are only referred to with the 'type' attribute. The binding simplifications outlined below in section 3.3.3 provide a more drastic solution to ensure the maintenance of the desired spec separation.

3.3.2.2 Binding Complexity: A Plethora of Options

It is evident that given the options detailed above, there are four main binding possibilities that toolkit vendors must support, each of which resulting in a different message format: RPC/encoded, document/literal, RPC/literal, document/encoded. As mentioned previously, RPC/encoded and document/literal are the bindings most commonly supported by existing toolkits. The need for RPC/literal and document/encoded endpoints is far less apparent. The main problem in the former case is that object references and arrays, constructs commonly used in remote procedure calls, are not inherently supported in XML schema; RPC calls are thus difficult to make without an appropriate encoding mechanism. In the second case, document/encoded bindings do not seem to serve any compelling purpose; in fact, we could not find an existing implementation supporting this type of endpoint. To complicate matters even further, additional message permutations are possible based on both the <wsdl:part> type-referencing mechanism and the <soap:body> 'encodingStyle'. In addition to having to support various encoding schemes, vendors must also deal with differing serializations for each style. The complexity of these binding options coupled with the fact that the spec remains silent with respect to their meaning and importance has led to a number of serious interoperability problems.

Firstly, toolkits are often incompatible for they support different types of bindings; for example, the current version of IBM Websphere Studio creates services that use RPC/literal endpoints while Microsoft ASP.NET does not. Moreover, as the current focus appears to be on providing remote procedure invocation, there are a large number of toolkits that do not support document bindings at all. The fact that the two software companies who drive industry standards cannot agree on a common set of functionality can only lead to the conclusion that the WSDL specification is far too ambiguous.

In addition, vendors have begun to use bindings improperly by exploiting the specification that they are only meant to detail data format. Microsoft's original SOAP toolkit implementation, designed to expose COM objects, generates RPC/encoded bindings by default. Contrarily, ASP.NET currently uses document/literal endpoints by default to describe remote procedure calls, which from a logical standpoint, is clearly incorrect. In taking this approach, they are not only creating their own arbitrary literal schema definitions, but they are also relegating XML document processing. A client using a toolkit which interprets the WSDL binding definitions properly would have a great deal of difficulty consuming a remote method created with ASP.NET. Recognizing document bindings, her toolkit would not generate any serialization code, thus forcing her to create an xml document by hand to represent the method call. Though it is evident that this scenario poses a rather large interoperability problem, it does raise an interesting point regarding the actual necessity of the different binding options.

These two issues introduce a subtler problem for the web services developer. In the vast majority of cases, WSDL is automatically generated from their source code, thus shielding them from the underlying binding details. As a result, interoperability is difficult or impossible to achieve for developers aren't familiar with WSDL, don't know what WSDL definitions their toolkits will generate by default, and don't know how to customize their toolkit's behavior. The situation outlined above involving the consumption of an ASP.NET service would indeed be very confusing to a developer who isn't aware of the RPC/document binding distinction. Moreover, one can imagine a whole host of other scenarios where developers are simply left to ponder why their endpoints aren't working properly.

Finally, in the distributed object technologies of old, such as CORBA, once a client was written, it could use the same proxy code to invoke objects in different servers; all it needed was the appropriate object reference. In the web services' world, with the way existing toolkits manipulate the bindings, there could be two identical operations residing on two different servers with different endpoint settings; one could be document/literal while the other is RPC/encoded. The client proxy code generated in one case will not be compatible with the other, thus drastically reducing service interoperability. Furthermore, although a WSDL description is currently generated from server-side code, it is not ensured that the server will remain compatible with that original definition after it is deployed. The versioning problem prevalent in existing component technologies is therefore compounded; the client must not only be recompiled when the server implementation changes, but also when the binding options change. As a result, it is evident that the existing specification precludes the use standard reusable interfaces for specific problem domains; e.g. health care, telecom, manufacturing etc.

3.3.2.3 SOAP Encoding: The Band-Aid Solution

Though they did leave the door open to other alternatives, the WSDL authors understood that XML Schema was a logical choice for describing SOAP messages. They realized, however, that in order to accurately represent remote procedure calls, there were already a number of existing toolkits that had implemented the SOAP encoding scheme. XSD did and still does not have implicit support for both arrays and object references, thus essentially leaving the WSDL caught between two seemingly imperfect standards for XML type representation. Despite the heavy criticism and the warnings against SOAP encoding issued by the original SOAP framers themselves, the WSDL authors still decided to leverage both specifications. As a result, there are essentially two serialization layers currently in use; WSDL provides a set of "base mapping" rules to generate an abstract schema definition based upon programmatic types and upon the specification of encoded bindings, the SOAP encoding rules provide the necessary capabilities for arrays and typed references to transform this definition into a series of concrete messages. This solution likely stems from the desire to support non-xml serializations; it was thought that specific encoding rules could still be applied to an abstract schema definition in these scenarios. However, the application of this methodology to the SOAP binding, as we will examine below, is completely illogical.

First and foremost, WSDL's set of base mapping rules, which represent the sole information source for the generation of XML Schema from native types, are glaringly incomplete. As mentioned previously in section 3.2.2.1, not only is it only "recommended" that they are applied, but the four generic rules provided in the specification do not even begin to address any core serialization issues. The lack of mapping standards is even more shocking in this case, for even though it is extraordinarily expressive, actually understanding XML Schema requires a significant amount of effort. The soapbuilders group can undoubtedly attest to the fact that even the manipulation of XSD's primitive types can be a daunting task. It is therefore clear that since toolkit developers are essentially forced to derive their own language-XSD mappings, inter-vendor interoperability is a virtual impossibility. The problem is in fact twofold; vendors are not only implementing mappings differently, but they are also supporting varying subsets of functionality. As there is currently no accepted standard, some toolkits only provide mappings for primitive types, while others provide more robust implementations incorporating arrays, enumerations, structs, classes etc.

Furthermore, the idealistic notion of incorporating two independent serialization layers is not fully realized in WSDL. Not only does the spec recommend that the SOAP encoding Array construct should be appropriately utilized in the abstract definition, but in order to accurately reflect native object references, the generation of the serialization code is a complex, undocumented venture involving the application of the encoding scheme to the base mapping rules. In fact, this task is sufficiently daunting to cause the vast majority of toolkits to use separate code paths for literal and encoded bindings; in essence, implementing entirely separate marshalling layers for each case. As one delves deeper into the underlying web services' plumbing, it becomes increasingly evident that SOAP encoding was a mere afterthought, designed as a temporary stop-gap measure until XML Schema was completed.

As mentioned above, the current serialization paradigm dictates that the SOAP encoding rules are applied to the abstract definition to produce concrete messages for passage over the wire. These encoding rules, however, are designed for application to a directed edge-labeled graph of nodes akin to the SOAP data model. It is unclear precisely how they are to be applied to a tree-based schema document, thus exposing a fundamental problem to the current encoding methodology. Both the SOAP and WSDL specifications remain silent on this issue; thereby creating further interoperability difficulties for this undefined task is also left to toolkit developers. As depicted below, the heart of the problem lies in the fact XML Schema is based on the Infoset and is thus incapable of implicitly representing object references; schema documents are therefore incapable of validating any serialized data structure containing references. The ensuing WSDL document applies the newly defined abstract syntax to describe a service, exposed with RPC/encoded bindings, for comparing two node objects. It can be noted that this example is kept rather simple for illustrative purposes; however, one could easily envision the serialization of more complex object graphs.

```
<definitions targetNamespace="http://www.example.com/node/wsdl"
             xmlns:noxsd="http://www.example.com/node/xsd"
             xmlns:tns="http://www.example.com/node/wsdl"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
```

```
               xmlns=”http://schemas.xmlsoap.org/wsdl”>

     <types>
          <schema targetNamespace=”http://www.example.com/node/xsd”
                    xmlns=”http://www.w3c.org/2001/XMLSchema”>

               <element name=”IsEqualRequest”>
                    <complexType>
                         <sequence>
                              <element name=”FirstNode” type=”noxsd:Node” />
                              <element name=”SecondNode” type=”noxsd:Node” />
                         </sequence>
                     </complexType>
               </element>

               <element name=”IsEqualResponse”>
                    <complexType>
                         <sequence>
                              <element name=”equality” type=”boolean” />
                         </sequence>
                     </complexType>
               </element>

               <complexType name=”Node”>
                    <sequence>
                         <element name=”name” type=”string” />
                         <element name=”data” type=”int” />
                    </sequence>
               </complexType>
          </schema>
     </types>

     <portType name=”IsEqualPortType”>
          <operation name=”IsEqual”>
               <input element=”tns:IsEqualRequest” />
               <output element=”tns:isEqualResponse” />
          </operation>
     </portType>

     <binding name=”IsEqualBinding” type=”IsEqualPortType”>
          <soap:binding style=”RPC” transport=”http://schemas.xmlsoap.org/soap/http” />
          <operation name=”IsEqual”>
               <soap:operation soapAction=”http://www.example.com/IsEqual” />
               <input>
                    <soap:body use=”encoded” encodingStyle=
                         “http://www.w3.org/2002/12/soap-encoding”>
               </input>
               <output>
                    <soap:body use=”encoded” encodingStyle=
                         “http://www.w3.org/2002/12/soap-encoding”>
               </output>
          </operation>
     </binding>
</definitions>
```

Figure 3.10 WSDL definition of ‘NodeComparison’ endpoint

Given this description, the following request message could be sent over the wire to invoke the service with two different node objects.

```
<env:Envelope xmlns:env="http://www.w3.org/2002/12/soap-envelope">
    <env:Body env:encodingStyle="http://www.w3c.org/2002/12/soap-encoding">
        <no:IsEqual xmlns:no="http://www.example.com/node/xsd">
            <no:FirstNode>
                <name>TestNode</name>
                <data>5</data>
            <no:/FirstNode>
            <no:SecondNode>
                <name>TestNode</name>
                <data>5</data>
            <no:/SecondNode>
        <no:/IsEqual>
    </env:Body>
</env:Envelope>
```

Figure 3.11 Single reference SOAP message for 'NodeComparison' endpoint

It would appear that SOAP encoding has produced node instances that do indeed match the schema definition for the <noxsd:Node> construct in the WSDL document. However, if SOAP encoding is based upon this SOAP data model and the SOAP data model does not use <xsd:complexTypes> to describe structured data, it is surely not wise to conclude that the two nodes are SOAP encoded instances of type <noxsd:Node>. Consider the SOAP message sent when the same node object is sent to the service.

```
<env:Envelope xmlns:env="http://www.w3.org/2002/12/soap-envelope">
    <env:Body env:encodingStyle=" http://www.w3c.org/2002/12/soap-encoding">
        <no:IsEqual xmlns:no="http://www.example.com/node/xsd">
            <no:FirstNode href="1" />
            <no:SecondNode href="1" />
        </no:IsEqual>
        <no:Node id="1" xmlns:no="http://www.example.com/node/xsd">
            <name>TestNode</name>
            <data>5</data>
        <no:/Node>
    </env:Body>
</env:Envelope>
```

Figure 3.12 Multi-reference SOAP message for 'NodeComparison' endpoint

In this case, the SOAP encoding multi-referencing mechanism has been used and it is evident that the node instances do not even come close to matching the schema definition. Not only do they not contain the specified child elements, but they also each have an undefined 'href' attribute.

It is clear that the cases where <noxsd:FirstNode> and <noxsd:SecondNode> do not resemble serialized instances of <noxsd:Node> can be directly attributed to XSD's inability to identify references between nodes. This shortcoming results in <wsdl:types> schema definitions that are incapable of validating the variety of possible data structure serializations resulting from the application of the SOAP encoding rules. WSDL documents specifying encoded bindings are essentially providing a schema to which all the resulting SOAP messages do not have to conform; a completely backward concept which virtually defeats the purpose of using XSD altogether. As a result, having first generated an abstract schema definition from a set of incomplete rules, toolkits must then somehow apply encoding rules to it to create messages, which cannot be properly validated. The implementation of a server that ensures that all incoming messages are correct is therefore extremely difficult, possibly even yielding unpredictable results. It is no wonder that even simplistic cross vendor toolkit interoperation is still but an abstract notion as yet unbound to a concrete reality.

In addition to the problems encountered with object reference representation, the serialization of arrays has also proven to be a daunting task. Firstly, the WSDL 1.1 spec merely 'recommends' the use of SOAP encoding Arrays, essentially allowing vendors to create proprietary mappings. Furthermore, as exemplified in the ensuing document sample, the use of this encoding Array construct is exceedingly convoluted. Figure 3.13 portrays the XML Schema representation of the following java statement, where the 'Result' type is a class with public 'Name' and 'Date' members.

```
Result ArrayOfResults[4];
```

```
<definitions targetNamespace="http://www.example.com/array/wsdl"
              xmlns:tns="http://www.example.com/array/wsdl"
              xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
              xmlns="http://schemas.xmlsoap.org/wsdl">

   <types>
      <schema targetNamespace="http://www.example.com/array/xsd"
                xmlns="http://www.w3c.org/2001/XMLSchema">
```

```
            <complexType name=”ArrayOfResults”>
                <complexContent>
                    <restriction>
                        <any namespace=”##any” minOccurs=”0” maxOccurs=”unbounded”
                                                        processContents=”lax” />
                        <attribute name=”arrayType” ref=”soapenc:Array”
                                                  wsdl:arrayType=”tns:Result[4]” />
                    </restriction>
                </complexContent>
            </complexType>

            <complexType name=”Result”>
                <sequence>
                    <element name=”Name” type=”string” />
                    <element name=”Date” type=”date” />
                </sequence>
            </compexType>
            .
            .
            .
    </types>
    .
    .
    .
</definitions>
```

Figure 3.13 WSDL 1.1 abstract definition exemplifying array representation

The content model of the SOAP encoding Array construct, though defined to be entirely extensible, is strictly controlled by the ‘arrayType’; an attribute which dictates both the type and size (all dimensions) of the array. It can be observed above that the value space of the arrayType is set by restricting the original <soapenc:Array> type in a particular manner. Since XML Schema does not have a mechanism for specifying the default value of an attribute containing a <xsd:QName> value, WSDL has introduced the ‘arrayType’ attribute to serve this purpose. Considering arrays are a fundamental programming concept, one cannot help feel overwhelmed by this excessively verbose syntax. It is true, in theory, that a toolkit would be responsible for automatically generating and consuming this definition; however, the oft misplaced key to interoperability is simplicity.

3.3.2.4 Inferring Programming Model

The debate between messaging and RPC programming models has been raging for years; Don Box wrote a MSDN article likening it to “distinguishing atoms from molecules” [40]. The document-centric view of the world defines messages as the atom thus elevating them to first-class status. Message purists tout that RPC is but one possible message exchange pattern and that by viewing the world in terms of remote procedure calls, no other patterns are obvious or even possible. On the other hand, RPC supporters tend to view the entire operation as the atom. They claim that with support for asynchronous invocation and one-way operations, any system designed around a messaging paradigm could be just as easily devised with an RPC mechanism.

Attempting to appease all parties, the authors of the SOAP specification tarnished their clean messaging paradigm by devoting an entire section to the description of remote procedure calls. Consequently, this decision was propagated into the WSDL spec, manifesting itself as an RPC/document binding distinction. In turn, toolkits currently infer the programming model from the message format defined in the WSDL document. For example, upon finding a <soap:binding> defining an RPC style, a toolkit will automatically generate all the appropriate serialization code. In this new world of openness and interoperability where service consumers have the freedom to choose both development platform and language, surely they should be given the opportunity to select a desired programming model as well.

This objective is indeed attainable for the truth of the matter is that the programming model and message format are completely orthogonal. In fact, as outlined earlier, ASP.NET's default functionality uses the RPC programming model to send and receive document style messages. Contrarily, it is also possible to send an RPC-style message using an XML API (e.g. DOM) and then dispatch the received response message to a routine for processing. Given that the examination of a SOAP message on the wire does not yield any information regarding the nature of the endpoint, it is apparent that the RPC/document distinction is indeed arbitrary. It simply serves to drastically complicate the bindings and thoroughly confuse both application and toolkit developers alike.

3.3.3 Interoperability Solutions

3.3.3.1 Removal of Encoded Binding Option

Given an object model, a schema compiler will generate both an XML Schema and the required instance documents; however, as alluded to earlier, the tree-based nature of the produced schema prohibits it from acting as an accurate template for all the possible serializations generated by the SOAP encoding rules. This shortcoming, coupled with a near non-existent programmatic type/XSD mapping specification, yields a virtually useless schema that merely provides clients with a proprietary hint to the service's pertinent structures. The underlying concept is that the application of a set of rules to *n* programmatic type systems to generate XML representations for data, which every toolkit can understand, is fundamentally flawed from an interoperability perspective. The creation of a single schema document to which all serializations must strictly adhere is a far superior approach. This goal can be achieved by the reconciliation of SOAP encoding and XML Schema, a task which will be undertaken in this section. This methodology would essentially remove the need for any encoding in the WSDL SOAP binding, thus drastically simplifying both the specification and toolkit implementations alike. The <wsdl:types> schema definitions would essentially always act as literal representations of the resulting wire format. Not only is schema portable and integrated with all other core XML technologies, but in defining a specific template for messages on the wire, it provides instant interoperability. Borrowing from the world of IDL, one could even envision forcing programmers to manually compose their schema definitions prior to writing a line of code. However, in reality, for fear of infuriating developers worldwide, this tedious task should be left to schema compilers. For this proposal to come to fruition, it is evident that a number of changes have to be instituted to the current serialization paradigm.

In order to generate an accurate schema from a programmatic type, the first issue to be dealt with is object reference representation. Being Infoset-based, element referencing is an impossibility in XML Schema, thus making the modeling of many common programmatic data structures rather difficult. The most drastic solution involves the creation of a new XML-based SOAP data model type system, which could describe, in unequivocal terms, families of graphs built according to the SOAP data model rules. Though very elegant and arguably the most 'correct', this solution requires the composition of an entirely new schema language from scratch, which in our estimation, is rather excessive. There is no doubt that XML schema will continue to represent a fundamental component of the XML technology stack for years to come. Being highly expressive, we believe that it can indeed be successfully leveraged to meet all the necessary requirements for the creation of robust web services' plumbing.

To accomplish this task, we propose a rather straightforward modification; instead of applying the SOAP encoding rules to the schema directly, toolkits could instead produce a schema describing the XML messages produced by these encoding rules. The following modifications to the <noxsd:Node> construct described in Figure 3.10 exemplify this idea.

```
<definitions targetNamespace="http://www.example.com/node/wsdl"
             xmlns:noxsd="http://www.example.com/node/xsd"
             xmlns:tns="http://www.example.com/node/wsdl"
             xmlns="http://schemas.xmlsoap.org/wsdl">

    <types>
        <schema targetNamespace="http://www.example.com/node/xsd"
                xmlns="http://www.w3c.org/2001/XMLSchema">

            <element name="IsEqual">
                <complexType>
                    <sequence>
                        <element name="FirstNode" type="noxsd:Node" />
                        <element name="SecondNode" type="noxsd:Node" />
                    </sequence>
                 </complexType>
            </element>

            <complexType name="Node">
                <sequence minOccurs="0">
                    <element name="name" type="string" />
                    <element name="data" type="int" />
                </sequence>
                <attribute name="id" type="ID" use="optional" />
                <attribute name="href" type="IDREF" use="optional">
            </complexType>
        </schema>
    </types>
    .
    .
    .
</definitions>
```

Figure 3. 14 WSDL <types> definition describing SOAP encoding serialization

It can be observed that in addition to making the element content optional, two attributes, 'id' and 'href', have also been included in the updated version of <noxsd:Node>. This new definition dictates that an <noxsd:Node> can either contain appropriate element content and an 'id' attribute or empty element content and a 'href' attribute; the former signifies an object instance while the latter represents a reference to that instance. These changes essentially allow the schema to validate all the possible message instances produced by the application of the SOAP encoding rules. By encapsulating the encoding referencing rules firmly into the schema document, the need for encoded bindings is essentially removed, thus immediately gaining the aforementioned benefits.

This solution, as outlined above, is not without its flaws. Firstly, every single type that could be used in a serialized graph would not only have to describe their element content as optional, but would also have to define equivalent 'id'and 'href' attributes with exactly the same semantics. This task would result in the creation of a huge amount of superfluous information, thus slowing processing time and making schema composition significantly more tedious. A more serious drawback, however, is that toolkits would have to assume that any type with an 'href' attribute intended it to be used for describing a graph. They would then be forced to filter out this encoding noise order to arrive at the pure data model. Finally, this resolution lacks any form of type checking support; though an XSD validator will match the values of the id and href attributes based on their types, the compatibility of the underlying constructs is not guaranteed.

The first two problems can be resolved by defining a pair of global attributes in a separate namespace and referencing them as needed. Though trivial to implement, in addition to leaving the type checking problems unaddressed, this solution fails to tackle the core type checking issue. Typed references are a fundamental building block of existing software development technologies and as such, XML Schema must be modified to support them synthetically. As many applications require that each element be uniquely identified, the <xsd:element> construct has a built-in 'id' attribute of type 'ID'. As a result, all that is required for reference support is the addition of an 'href' attribute of type 'IDREF' to the <xsd:element>. Validators could then easily be amended to perform type checking upon encountering this newly defined metadata. Proposing changes to the ratified XML Schema specification may not be fully realistic from a political perspective; however, if XSD is to fulfill the role of interoperable programming language intermediary, this modification is absolutely critical.

Having simplified the serialization of typed references, the elimination of the ambiguity surrounding XML array representation can now be undertaken. As outlined previously, not only does WSDL 1.1 not specify a definitive array mapping mechanism, but the SOAP encoding Array construct it does recommend, as exemplified in Figure 3.13, is both verbose and unintuitive. We believe, however, that native schema constructs are indeed expressive enough to provide a straightforward solution worthy of the creation of a de-facto industry standard. To illustrate, example 3.13 can be rewritten as follows:

```
<definitions targetNamespace="http://www.example.com/array/wsdl"
             xmlns:tns="http://www.example.com/array/wsdl"
             xmlns:arrxsd="http://www.example.com/array/xsd"
             xmlns="http://schemas.xmlsoap.org/wsdl">

    <types>
        <schema targetNamespace="http://www.example.com/array/xsd"
                xmlns="http://www.w3c.org/2001/XMLSchema">

            <complexType name="ArrayOfResults">
                <sequence>
                    <element name="Result" type="arrxsd:Result" minOccurs="0" maxOccurs="4"
                            nillable = "true" />
                </sequence>
            </complexType>

            <complexType name="Result">
                <sequence>
                    <element name="Name" type="string" />
                    <element name="Date" type="date" />
                </sequence>
            </compexType>
            .
            .
            .
    </types>
    .
    .
    .
</definitions>
```

Figure 3.15 Proposal for XSD array representation

It can be observed that in addition to providing comparable expressive power to the SOAP encoding Array, the direct manipulation of the <xsd:element> construct is syntactically far cleaner. There is no complex XSD derivation by restriction syntax and the size and type information previously acquired from the <wsdl:arrayType> attribute has been seamlessly encapsulated into the native schema 'element'. This document would not only be easier to decipher for potential clients perusing WSDL repositories, but the associated toolkit processing code would clearly be a great deal simpler. A perceived drawback of this approach is the necessity to create an additional wrapper construct of type

<arrxsd:ArrayOfResults> for multi-dimensional array representation. However, in the rare case that this scenario even occurs, we firmly believe that the new syntax is still far simpler and more expressive than that of the SOAP encoding schema. As a result, in the spirit of simplicity and interoperability, we propose that the use of the SOAP encoding Array be discontinued altogether in favour of the industry-wide adoption of the preceding method.

Having proposed that programmatic types can be very accurately mapped to a slightly revised version of XML Schema, one final question remains: how exactly will this mapping occur? As we've highlighted repeatedly, the WSDL spec only provides an optional, incomplete set of serialization guidelines, thus creating an interop disaster for vendors are prompted to formulate their own proprietary mappings. This situation is a far cry from the detailed per language mapping specs provided by the OMG for its CORBA technology [41]. We strongly believe that a similar body of work must be erected for XML Schema; the mapping of virtually all programmatic constructs, ranging from primitives to static members to abstract types, must be addressed in detail. It can be noted that this glaring omission from the web services' initiative has political roots; to date, the specification of language mappings, quite simply, has not been assigned to a W3C working group. In fact, the WSDL WG's mandate even explicitly states that they are not responsible for this task. As it represents the final core component required for the successful development of an 'internet' of interoperating applications, the publication of this complete, standard set of serialization rules must be immediately undertaken.

3.3.3.2 Towards a Standard Message Format

Having emphasized the importance of the removal of encoded bindings to ensure schema validation of a service's parameters, we will now broaden this concept to the message itself. In order to maximize interoperability, endpoints must have the ability to validate incoming messages at the level of the SOAP body. The elimination of the <wsdl:message> construct represented the necessary first step to this end while the consolidation of the RPC and document binding styles is the next.

As outlined previously, RPC and document bindings simply serve to artificially dictate SOAP message format. Since they do not actually have any bearing on the programming model adopted at the endpoints, this futile binding distinction can therefore be abolished.

This change represents the final step towards achieving the vision of a standardized, schema-verifiable SOAP message format. The original purchase order example has been modified to include all of our proposed changes.

```
<definitions targetNamespace=”http://www.example.com/PO/wsdl”
              xmlns:poxsd=”http://www.example.com/PO/xsd”
              xmlns=”http://schemas.xmlsoap.org/wsdl”>

    <types>
        <schema targetNamespace=”http://www.example.com/PO/xsd”
                  xmlns:tns=”http://www.example.com/PO/xsd”
                  xmlns=”http://www.w3c.org/2001/XMLSchema”>

            <element name=”ProcessPORequest”>
                <complexType>
                    <sequence>
                        <element name=”PO” type=”tns:PurchaseOrder”>
                        <element name=”Person” type=”tns:Person”>
                    </sequence>
                </complexType>
            </element>

             <complexType name=”PurchaseOrder”>
                <sequence>
                    <element name=”buyer” type=”string” />
                    <element name=”item” type=”string” />
                    <element name=”date” type=”date” />
                </sequence>
             </complexType>

            <complexType name=”Person”>
                <sequence>
                    <element name=”name” type=”string” />
                    <element name=”department” type=”string” />
                    <element name=”phoneExtension” type=”int”>
                </sequence>
            </complexType>
        </schema>
    </types>

    <portType name=”ProcessPOPortType”>
        <operation name=”ProcessPO”>
            <input element=”poxsd:ProcessPORequest” />
        </operation>
    </portType>

    <binding name=”ProcessPOBinding” type=”ProcessPOPortType”>
        <soap:binding transport=”http://schemas.xmlsoap.org/soap/http” />
        <operation name=”ProcessPO”>
            <soap:operation soapAction=”http://www.example.com/ProcessPO” />
            <input>
                <soap:body encodingStyle=  “http://www.w3.org/2002/12/soap-encoding”>
            </input>
        </operation>
    </binding>
</definitions>
```

Figure 3.16 WSDL definition reflecting complete set of proposed changes

The SOAP message format dictated by this definition is exemplified below.

```
<env:Envelope xmlns:env="http://www.w3.org/2002/12/soap-envelope">
    <env:Body env:encodingStyle=" http://www.w3c.org/2002/12/soap-encoding">
        <po:ProcessPORequest xmlns:po="http://www.example.com/PO/xsd">
            <PurchaseOrder>
                <buyer>Fictional Construction Ltd.</buyer>
                <item>Concrete</item>
                <date>01/11/2002</date>
            </PurchaseOrder>
            <Person>
                <name>Craig Mckinley</name>
                <department>Purchasing</department>
                <phoneExtension>4356</phoneExtension>
            </Person>
        <po:/ProcessPORequest>
    </env:Body>
</env:Envelope>
```

Figure 3.17 SOAP message invoking purchase order service

It can be observed that in addition to the elimination of the <wsdl:message> construct, both the 'style' and 'use' attributes have been removed from the <soap:binding> and <soap:body> tags respectively. With this methodology, the <wsdl:types> schema generated from the set of language specific mapping rules provides service consumers with a strict template for what must be present on the wire. Consequently, as depicted in Figure 3.18, the contents of all incoming SOAP messages can be validated by the 'ProcessPORequest' schema element; any ambiguity regarding the format of a 'correct' message is therefore removed. Moreover, the removal of the binding options precludes the need for the existing conditional, contradictory set of rules for generating SOAP messages, thus drastically simplifying the lives of both web services' developers and toolkit vendors alike. We are effectively left with a clear, concise definition, ultimately providing developers with the necessary tools to create perfectly interoperable endpoints.

Finally, it is interesting to note that this paradigm allows a service to be defined simply in terms of the messages exchanged; it is left up to the individual developers to map those messages to method call stacks if desired. Toolkits should, in essence, provide web service developers with the decision to create function or message oriented endpoints at WSDL compilation time; the 'Person' and 'PurchaseOrder' constructs could still either represent documents for processing or RPC parameters. Provided that the message format adheres to the schema, the selected approach is utterly irrelevant. One could even envision the scenario of endpoints communicating with different programming models, thus taking interoperability to a whole new level.

3.4 Conclusion

There is little doubt that the unnecessarily complex WSDL specification is responsible for the interoperability problems plaguing web services today; the situation is currently so dire that cross vendor toolkit communication for even the simplest of operations has proven to be a truly daunting task. In addition to being syntactically unclear, the description methodology employed by existing WSDL documents is fundamentally flawed. The spec currently outlines a host of elaborate, erroneous rules for manipulating WSDL constructs in order to generate the SOAP messages necessary for service invocation. The complexity of both these guidelines and the SOAP encoding rules have led to a situation wherein a message that is deemed correct by one toolkit will be rejected by another. It is evident that the most elegant solution to this dilemma is the definition of a template for precisely what must appear on the wire; an XML Schema definition for the SOAP message required for successful method invocation essentially guarantees instant interoperability. In order for this vision to become a reality, however, several key amendments to both WSDL and XSD must be implemented.

Firstly, once XSD is accepted as the de-facto standard for type representation, it can be leveraged to replace the superfluous <wsdl:message> construct. In addition to drastically cleaning up the WSDL abstract definition, this revision essentially paves the way for the simplification of the SOAP bindings. To this end, the removal of the RPC/document and encoded/literal distinctions serve to ease toolkit implementation by effectively precluding message format from being dictated by a complex set of SOAP generation guidelines. Alternatively, both the definition of a standard set of schema mapping rules for each programmatic type system as well as the addition of object reference support to XSD allow for the creation of an accurate schema describing an entire service call. Including this information directly into WSDL documents themselves effectively removes any message level ambiguity, thus drastically simplifying the underlying infrastructure and ensuring the future success of the web services paradigm.

Chapter 4
Enterprise Web Enablement

4.1 Introduction

Interoperability is by far the most compelling quality of the web services' revolution to date; distributed computing is poised to be forever transformed by the promise of a framework which allows networked code to seamlessly communicate regardless of hardware platform, operating system, implementation language and programming model. Having proposed the necessary fundamental amendments to WSDL to ensure that this lofty objective becomes a reality, we have undoubtedly cleared the path to successful interoperability amongst the sixty plus toolkits available on the market today. This newly recommended infrastructure can now be leveraged to address the fundamental problem introduced in the first chapter: the creation of a framework for the enterprise web enablement of legacy assets.

The migration of systems to the internet is a costly, complex venture for which an efficient generic solution has yet to be developed. This task essentially encompasses two distinct facets; the distribution of information to both interactive users via a thin client as well as to business partners via SOAP. As discussed in chapter two, the latter can be performed in some cases by existing web services toolkits, which automatically wrap existing business logic with a simplistic 'web' wrapper based upon provided interface definitions. This functionality is sufficient for the primitive services available today; e.g. weather updates, stock quotes, and numeric conversions. However, the creation of enterprise web applications that are reliable, highly available, fault-tolerant, and scalable has yet to be fully realized. In order for organizations worldwide to begin exposing their critical digital assets over HTTP, whether it be via SOAP or HTML, it is imperative that the task of transforming legacy applications into robust, enterprise level web installations be drastically simplified. We strongly believe that it will only be upon the heels of this realization that traditional data barriers will come crashing down, thus allowing the vital information driving industry today to easily flow from deep within the enterprise directly to clients and business partners.

In recent years, the term 'enterprise application' has been reserved for systems built with middleware technologies residing within the confines of an application server: J2EE's

Enterprise Java Beans, Microsoft's Component Object Model+, and the OMG's CORBA Component Model. Countless person-years have been invested into these technologies to supply application developers with robust, feature-rich, high performance runtime environments. As outlined in the second chapter, these platforms provide the vast majority of the underlying plumbing required in large scale web apps: from security, transactions, and clustering to lower level necessities such as threading, connection management, and instance pooling. As the duplication of this functionality would involve an enormous development effort, legacy web enablement strategies should clearly leverage these existing platforms. To this end, we now propose a reference architecture and supporting toolkit, which essentially serve to modernize legacy applications by wrapping them with robust middleware components. In this paradigm, legacy systems would automatically leverage the capabilities of their distributed object technology wrappers, thus allowing them to be seamlessly integrated into a secure, transactional, scalable, fault-tolerant web environment. As outlined in the second chapter, there have been a variety of differing strategies applied to bridge the gap between legacy and middleware code; we will now propose, however, that existing XML web services infrastructure should be leveraged to accomplish this task.

4.2 Reference Architecture

4.2.1 Overview

The data-centric, platform neutral nature of XML web services has the vast majority of the foremost technology players frantically re-positioning their product strategies. Despite all the excitement, however, there is still a great deal of uncertainty regarding the actual application of this bleeding-edge technology. The notion has been put forth that web services will make service-oriented architectures feasible on a large scale, thus effectively extending the object oriented paradigm to create an internet of reusable components. The more popular belief, however, is that web services will serve as the glue for inter-enterprise integration. They will essentially act as 'middleware for middleware', successfully bridging the gap between the proprietary technologies employed by different companies. In this chapter, however, we explore the idea of pushing web services from the periphery directly to the heart of enterprise installations. As depicted below in Figure 4.1, our proposed reference architecture utilizes XML as the underlying communication mechanism between legacy systems and a layer of middleware component wrappers.

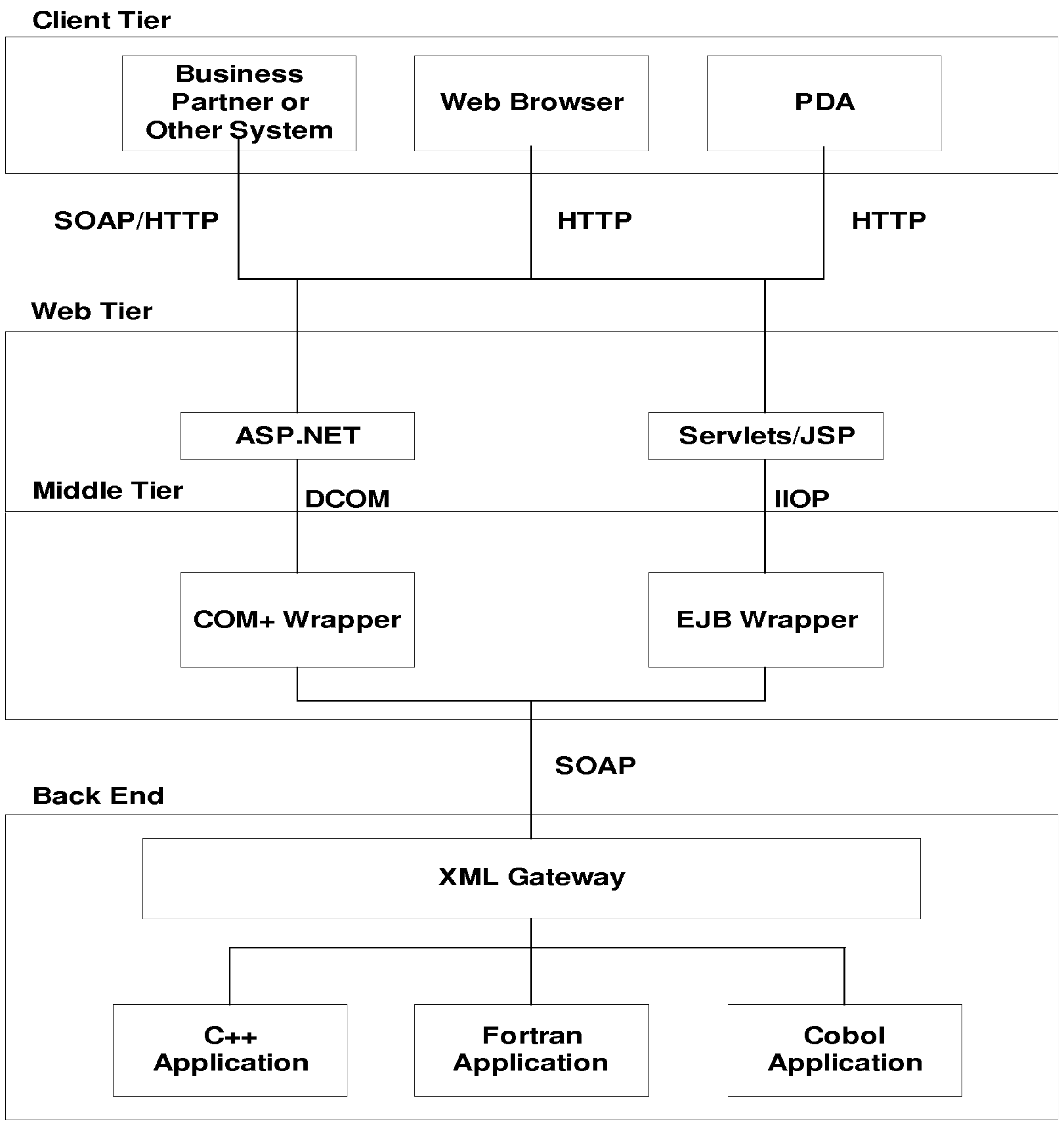


Figure 4. 1 Reference Architecture for Enterprise Web Enablement

It can be observed that legacy systems are assumed to reside on back end machines behind an XML gateway, which simply represents an abstraction for the plumbing associated with the web service paradigm: stubs, skeletons, multi-threaded SOAP processor, and (de)serializers. These systems can be effectively XML-enabled in a non-invasive manner, thus allowing them to successfully consume standardized messages produced by the middle tier. Not only can these applications be easily extended with the addition of code to the middle tier, but the distributed object technologies also serve to add robust security, transaction, and clustering functionality. Furthermore, as we will explore in the ensuing chapter, by allowing a multitude of disparate systems to seamlessly communicate, the middleware automatically provides enterprises with an integration platform. Finally, exposing the middle layer via a web tier allows the underlying business logic to be readily available to applications worldwide via SOAP or to interactive clients using a web browser. As outlined below in the clustering section, though each of the layers could well reside on separate machines, the most effective solution groups the middle and web tiers together and pushes the legacy implementation(s) to separate host(s).

4.2.2 The Power of XML

Though the proposed architecture clearly presents a plethora of interesting benefits, the true extent of which will be detailed throughout the remainder of this paper, its most compelling feature is the use of an XML gateway to integrate the legacy systems. Unlike the solutions discussed in the second chapter (e.g. adapters, CORBA etc.), which mandate the use of proprietary interfaces, the application of XML shifts the focus to standardized data transfer; it effectively places data, not code, at the center of the computing universe. The legacy systems and middleware tier are merely required to produce and consume standardized XML messages, thus giving the architecture a huge amount of flexibility and extensibility by effectively decoupling the two layers. Not only can legacy code written in any language be wrapped with virtually every middleware technology, but either tier can also be replaced given that the alternative understands the appropriate set of data messages.

Furthermore, integration projects of the past normally required vastly differing solutions; for example, C++ applications could be exposed as a set of CORBA components while a CICS system could be coerced into communicating with an EJB wrapper via a proprietary adapter. The web service paradigm effectively represents a silver bullet for it offers a clean solution to virtually every integration problem. Unlike existing distributed object technology protocols (DCOM, IIOP), web services standards are fully open and can be mapped to virtually every existing technology; it is no wonder that they are rapidly amassing industry-wide support. SOAP and WSDL do truly represent the glue which will serve to transform the existing heterogeneous computing environment filled with countless disparate technologies into a homogeneous world of standardized data transfer. The XML enablement of legacy assets therefore guarantees their survival for years to come, thus saving corporations millions in integration and redevelopment costs. Finally, the future ubiquity of web services is further guaranteed by the fact that development of the necessary infrastructure is significantly simpler than that of the aforementioned integration alternatives.

The overwhelming advantages of leveraging XML data transfer as a means of integration do, however, come at the price of performance. Firstly, as XML uses plaintext tags to describe each piece of information, SOAP creates a significant amount of network traffic compared to its binary counterparts. As a result, the deployment of the middleware wrapper to a separate machine does indeed serve to create a noteworthy performance burden. This inefficiency could be improved in certain scenarios by placing the middle tier on the same host as the legacy system. In a trivial solution, they could communicate via an HTTP loopback mechanism; however, a more sophisticated implementation would involve the passage of SOAP messages over the native RPC mechanism of the underlying operating system (e.g. windows messaging). The reality, however, is that in the vast majority of situations, the integration potential coupled with the simplicity, flexibility, and extensibility of the proposed distributed architecture far outweigh any performance concerns associated with XML communication over an additional LAN network hop. It can be noted that in addition to the increased bandwidth, the processing of the XML messages at each network endpoint incurs additional time delays. The truth of the matter, however, is that in most cases, this additional time will be dwarfed by network communication, I/O access, as well as the execution of the actual heavyweight business logic. Furthermore, the overhead will be minimized as XML technology continues to evolve; streaming SAX and pull parsers have indeed proven to be increasingly efficient compared to their DOM predecessors.

4.2.3 Deployment

As we are attempting to provide an efficient, cost-effective, maintainable method to web-enable legacy systems, it is imperative that the realization of this proposed architecture be exceedingly straightforward. Despite the best efforts of application server vendors, development using current middleware technologies remains overwhelmingly complex. There is little doubt that mastering the intricate subtleties of either COM+ or EJB for successful enterprise development is a daunting task well beyond the capabilities of the majority of programmers. Fortunately, however, it is indeed possible to automate this development process; as a result, we propose the creation of an 'Enterprise Web Enablement' (EWE) toolkit, which essentially extends the functionality of existing web services toolkits to generate both a middleware wrapper and web tier. As exemplified below in Figure 4.2, in order to accomplish this task, our proposed toolkit requires both a legacy system interface as well as a variety of configuration information.

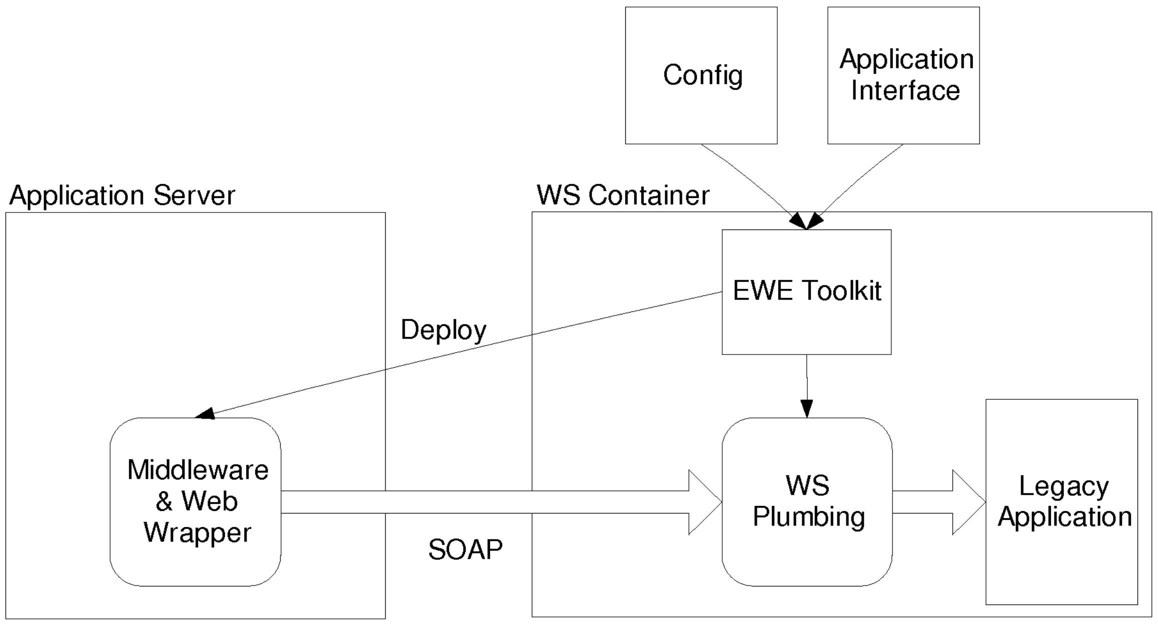


Figure 4. 2 EWE Toolkit Deployment Strategy

Having decided on the type of application to be deployed at the application server, the developer is then given the option to dictate precisely how it is to be generated. In the scenario that logic is to be added to the middleware layer, the toolkit could be configured to produce all the relevant source code; the more common scenario, however, would involve the generation of a single deployment file. This file could then either be deployed to the application server manually or, alternatively, as application servers offer programmatic interfaces, the toolkit could automatically deploy it remotely. As a result, having installed an app server, legacy application, and EWE toolkit, a developer need only allow the toolkit to process the set of desired interfaces in order to produce an entire enterprise web environment.

Later in the chapter, we will outline how with only a minimal amount of additional configuration, the legacy system can leverage the robust services offered by the EJB container.

4.3 Leveraging Middleware Functionality

4.3.1 Component Wrapper Generation

4.3.1.1 Overview

Having provided a brief overview of the proposed reference architecture and deployment strategy, the specific wrapping methodology can now be specified. In order to present precise, tangible explanations of the proposed concepts, we will be using Enterprise Java Beans technology in all the examples for the remainder of the paper. It can be noted, however, that although the underlying implementation may differ slightly, unless otherwise specified, all of the ideas touched upon map directly to COM+.

The specifics of the proposed functionality are best illustrated with a trivial example. Let us assume that a bank has a legacy C++ application that is used internally by tellers during their interaction with customers. It would clearly be highly beneficial to allow business partners to leverage this system for billing purposes by exposing it as a web service. The application, represented by the header files below, implements simplistic banking logic capable only of withdrawing and depositing funds to a specified account.

```
#include Account.h

public class Banking
{
public:
    Banking();
    int withdraw(Account i_objAccount, double m_dAmount);
    int deposit(Account i_objAccount, double m_dAmount);
}
```

Figure 4. 3 Banking interface

```
public class Account
{
private:
    int m_iType;
    int m_iAccountNumber;
    int m_iOverdraftLimit;
    char* i_strName;
public:
    Account(int i_iType, i_iOverdraftLimit);
    public int getType();
    public int getAccountNumber();
    public int getOverdraftLimit();
    public char* getName();
}
```
Figure 4. 4 Account interface

Identifying 'Banking.h' as the application's external interface, the header files along with a variety of configuration information (e.g. desired endpoint location) are passed to the EWE toolkit for processing. The result is the generation of all the elements required to web enable the application in an enterprise fashion. The processing carried out by the toolkit can essentially be divided into three separate tasks: making the banking application callable over SOAP, generating the middleware wrapper to invoke it, and finally exposing the wrapper via the web tier.

The installation on the machine hosting the banking application, depicted on the right below in Figure 4.5, is essentially identical to those on existing web service servers described previously in chapter two. The generated dispatcher object spawns a new thread to process and route each incoming SOAP request to the appropriate skeleton. This proxy object is then responsible for performing the appropriate (de)serialization of any complex types (e.g. Account) and invoking the legacy application. Tooling is currently sophisticated enough that in the case of dynamic languages such as java and C#, code can be web service-enabled by simply deploying applications into the web services container. However, the majority of older programming languages, C++ included, do not have runtimes and consequently do not permit the deployment of object files to a distinct server process. As a result, the developer is forced to compose a simple server driver program responsible for registering the application with the web services container. This program, along with the generated dispatcher, skeleton, (de)serializers, and the legacy implementation must then be explicitly compiled and linked before starting the server. It can be noted that in cases where it is infeasible to recompile the legacy code, the XML infrastructure code can first be compiled separately and then appropriately linked in.

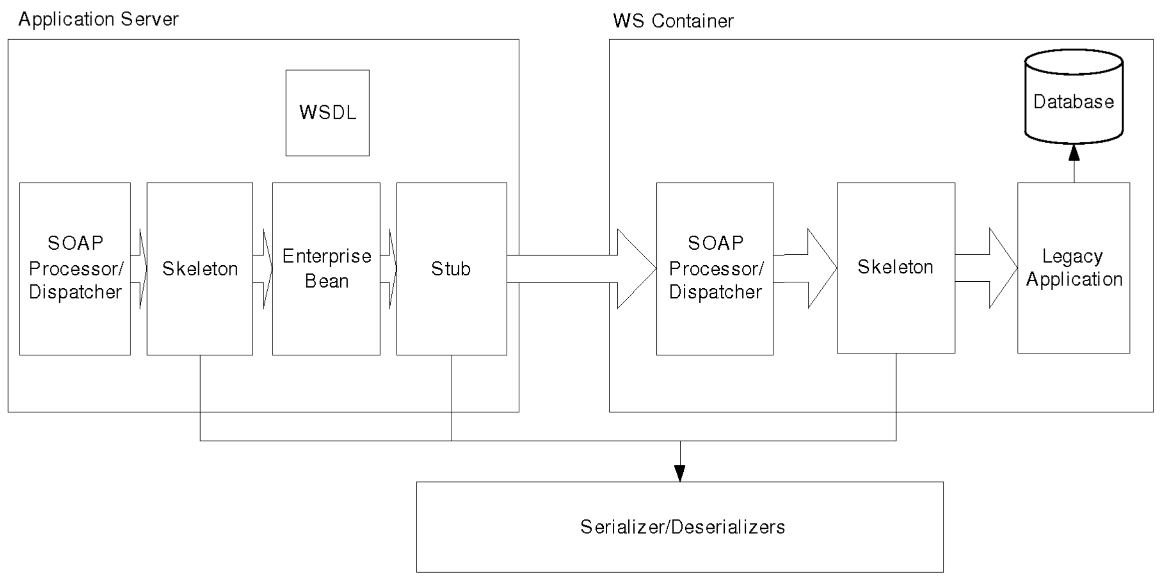


Figure 4. 5 Detailed EWE Reference Architecture

In order to complete the creation of the enterprise web system, the EWE toolkit must also produce the necessary wrapper components for deployment into an application server. As exemplified above, this task involves the generation of a stateless session enterprise java bean wrapper, the stub and (de)serializers required for communication with the C++ implementation, as well as the specified web tier modules. It can be noted that in addition to the actual bean implementation, the creation of the EJB requires the generation of home, remote, and service endpoint interfaces as well as a deployment descriptor. Depending upon the configuration information passed to the tool at deployment time, it will either expose the middle tier to business partners via SOAP, as depicted above, or to interactive web clients via HTML pages. In the former case, in addition to a dispatcher, skeleton, and (de)serializers, a WSDL file describing the EJB wrapper is generated for publication to a registry. In the latter case, however, an HTML page and servlet are generated according to the model view controller (MVC) design pattern [42].

The nature of the web tier created will depend on both the type and desired use of the web-enabled legacy application. The banking application could, for example, be exposed using either method; a browser interface would offer clients direct access to their accounts from the bank's website, while a SOAP interface could provide merchants partnering with the bank with a programmatic interface with which to charge their clients.

In the former case, it can be noted that the HTML pages produced by the EWE tool would be very rudimentary. XSLT transforms would simply be used to produce HTML input mechanisms based upon the input parameters described in the legacy WSDL document [43]. For example, a string parameter would produce a large textbox, a numerical input would produce a smaller textbox, an enumeration would produce a combo box etc. The Account object depicted above would simply output a group of textboxes for all of the constituent members of the Account class. In more complex scenarios, involving a large number of components, one could envision the generation of a complete front end using a template-based configuration mechanism; this task is, however, beyond the scope of this paper. Furthermore, it can be noted that in some cases, as legacy systems may already have some form of user interface, it would indeed be possible to perform user interface migration. The research in this area has been extensive to date, producing a number of methodologies and tools to carry out this task automatically [22].

4.3.1.2 The Messaging Alternative

In addition to raging on over SOAP message format as discussed at length in chapter three, the RPC/messaging debate continues into the depths of the implementing middleware technology. The vast majority of existing web services toolkits embrace the RPC model, thus primarily promoting synchronous communication between the client and server. However, in providing an asynchronous programming model and guaranteed message delivery, messaging systems are undoubtedly fitting to a world of unpredictable network latencies, long-running services, and server unavailability. They would seem to offer a graceful solution to the concerns that HTTP is not sufficiently reliable for mission-critical services; though IBM has put forth a proposal to meet this challenge (HTTPR), its widespread adoption is unlikely at best [44]. There is indeed little doubt that the loosely-coupled, message-oriented nature of web services maps well to messaging systems, thus inferring that the EWE toolkit should offer the option to generate message beans or COM+ queued components. It should be noted that at the time of writing, message beans do not seamlessly support asynchronous communication; however, it is a feature expected in the next draft of the EJB spec. These components are identical to their RPC counterparts with the exception that they can only be invoked from a messaging middleware implementation (JMS or MSMQ respectively).

In this type of environment, web services clients would use an API very similar to that of the RPC model. In this scenario, however, processing can continue immediately after service invocation and as messages are persisted to the middleware, their delivery is guaranteed. Though toolkits supporting this type of functionality are a rarity, their popularity is gaining momentum; Apache Axis announced their intention to support JMS by summer 2004. However, until tooling matures to the point that messages can seamlessly flow between differing MOM implementations, this type of architecture will remain confined to the white board. Moreover, the largest deterrent to the adoption messaging-driven web services is the support for asynchronous invocation and guaranteed delivery that RPC toolkit vendors are building into their products. The compelling advantages of the messaging paradigm are therefore mitigated, leading us to the belief that the generation of stateless RPC components is indeed sufficient.

4.3.1.3 Coarse-Grained Interfaces

While the simple banking application outlined above is represented by a single external interface, in reality, legacy applications may well expose numerous components. In this scenario, the toolkit is passed the appropriate set of interfaces and will, by default, preserve the granularity of the implementation by producing an EJB wrapper for each component. Though desirable in certain situations, this technique may result in a huge amount of deployed middleware code, thus complicating the system and posing a maintenance problem. As a result, the EWE toolkit should adhere to one of the salient features of the service-oriented paradigm: the creation of coarse-grained interfaces. By offering the option to create a single wrapper for a multitude of legacy components, the middleware effectively acts like a distributed façade, thus providing the client with a higher level view of the back end implementation. For example, let us assume that the legacy application also contains a CreditCard component, which when combined with the Banking module, can create a more complete ‘Financial’ service.

```
public class CreditCard
{
public:
    CreditCard();
    void payBill(int i_iCardNo, float i_fAmount);
    char* getBillDueDate(int i_iCardNo);
    float getBillBalance(int i_iCardNo);
    void charge(int i_iCardNo) throws InsufficientCreditException;
}
```

Figure 4. 6 CreditCard interface

The toolkit could be configured to simply generate a single Financial EJB component comprised of all the operations from its two constituent interfaces. When exposed as a web service, the generated WSDL file would only have a single portType, making it appear as though a single service offers the capability to pay credit card bills with the funds withdrawn from a specified account. It is clear that permitting the middleware to call a fine-grained set of components not only reduces the amount of code deployed to the application server, but also drastically simplifies the ease of use and maintainability of the web installation.

4.3.2 Security

4.3.2.1 Introduction

Permitting access to mission critical systems from beyond the firewall is clearly an unsettling prospect for IT managers worldwide. As security has, however, been a huge concern since the advent of the internet, mechanisms have been successfully implemented at length for browser clients. The combination of HTTP authentication and secure sockets layer (SSL) has indeed proven to be very effective. However, the notion of SOAP messages tunneling through port 80 to access legacy assets has introduced new security requirements, the true extent of which have yet to be fully realized. As a result, this section will focus predominantly on addressing the security concerns when exposing legacy systems as web services. By leveraging both emerging standards and existing middleware functionality, we strongly believe that it is indeed possible to create robust, secure installations. Furthermore, given that the majority of legacy systems are not identity-aware, the necessary security mechanisms can be automatically generated based simply upon configuration information provided at deployment time. This paradigm effectively creates a clear separation with the business logic, thus leaving the authentication, authorization, integrity, confidentiality, and non-repudiation characteristics of the system to the platform.

4.3.2.2 Background

Departure from SSL

Web services are currently secured with the use of SSL and HTTP basic authentication; however, despite being accepted as a robust industry standard for browser clients, this solution severely limits the flexibility of the web services paradigm. Firstly, whereas SSL is a point-to-point mechanism inherently tied to the transport layer, SOAP messaging can, by definition, include intermediaries belonging to organizations other than the service requestor or provider. Consequently, assuming any of these third parties are even capable of reading the messages, decryption and re-encryption of the data at each entity is far too much of a performance burden. Furthermore, in many cases, users may only wish to encrypt certain portions of a message, a capability not available in SSL. Finally, the use of transient session keys makes non-repudiation a true impossibility; in case of a dispute, neither party can undeniably claim to own the unaltered message. Fortunately, there is a viable solution to these problems as standards for embedding security information directly into the SOAP header are slowly beginning to emerge. Placing security at the message level provides a n umber of compelling benefits as requests are permitted to securely traverse through multiple network layers, topologies and intermediaries independent of the underlying protocol.

WS-Security

Completed in April of 2002 by Microsoft, IBM, and VeriSign, the WS-Security specification provides a means of encapsulating functionality from the XML Encryption and XML signature specifications directly into a SOAP message [45, 46]. As the names imply, XML signature provides a means of incorporating digital signatures into XML documents while the XML encryption outlines a schema for encrypted data. By operating at varying levels of granularity, these specs offer a huge amount of flexibility. For example, in many workflow scenarios where an XML document flows stepwise between participants and a digital signature implies some sort of commitment or assertion, each participant may wish to sign only that portion for which they are responsible and assume a concomitant level of liability. The additional flexibility is also critical in scenarios where it is important to ensure the integrity of only certain portions of an XML document, while leaving open the possibility for other portions of the document to change. A signed XML form delivered to a user for completion is a perfect illustration of this scenario. Older standards for digital signatures did not provide either the syntax for capturing this sort of high-granularity signature or the mechanisms for expressing which portion a principal wishes to sign.

In addition, the WS-Security spec also provides a means of including identity credentials, such as a username/password pair, certificate (X.509), Kerberos ticket or SAML assertion, in the SOAP header [48]. These credentials, coupled with a digital signature, provide integrity, authentication, and non-repudiation while the encryption standard provides confidentiality. As we will discuss the below, the integration of these technologies into our reference architecture will permit services to leverage the role-based authorization mechanism of the middleware platform, thus fulfilling the standard set of security requirements.

Role-Based Access Control

Both EJB and COM+ environments implement a role-based access control security model, which essentially maps entities to roles and then roles to resources. The middleware platform allows these security policies to be defined declaratively down to the granularity of the method, thus effectively delegating authorization to the container. In the java environment, J2EE's portable security API, the Java Authentication and Authorization Service (JAAS) is leveraged [49]. The true power of JAAS lies in its ability to use virtually any underlying security system. The developer must simply provide a custom login module, serving to authenticate users and associate them with a security context, to plug into the JAAS framework. This generated context is then automatically appended to calls to the EJB container, at which time authorization based upon the provided security policy is performed.

Single Sign-on

In an open e-business world, many unfamiliar clients will need to gain access to web services; at a certain point, however, it will become rather inefficient to setup security information (e.g. usernames and passwords) in the application server for every possible identity. As a result, the future of web service authentication lies with the notion of single sign-on. Current projects such as Microsoft Passport and Liberty Alliance represent the first generation of these single sign-on services, where users need only authenticate once at a centralized authority in order to gain access to a set of web services. The Security Assertion Markup Language (SAML) attempts to standardize this concept by defining an XML-based framework for exchanging security information. In addition to defining precisely how the security credentials, or assertions, are represented in XML, SAML also outlines the message exchange protocol for querying the authority service. The basic premise is that in exchange for appropriate authentication information, a user obtains a security assertion from a SAML authority. This assertion could then be rapidly accepted without formal authentication by a host of services within the domain of trust. The standardization of security assertions effectively alleviates corporations from managing the authentication information of all their business partners, thus ultimately leading to an open-ended, more flexible, more scaleable e-business infrastructure.

4.3.2.3 Securing the Reference Architecture

It is evident that many web services, such as those in the banking domain depicted above, will require robust security mechanisms fulfilling each of the five standard security requirements: authentication, authorization, confidentiality, integrity, and non-repudiation. Based on the above discussion, it is clear that incorporating the functionality of the standardized WS-Security SOAP header into our reference architecture, depicted below in Figure 4.7, will fulfill these needs.

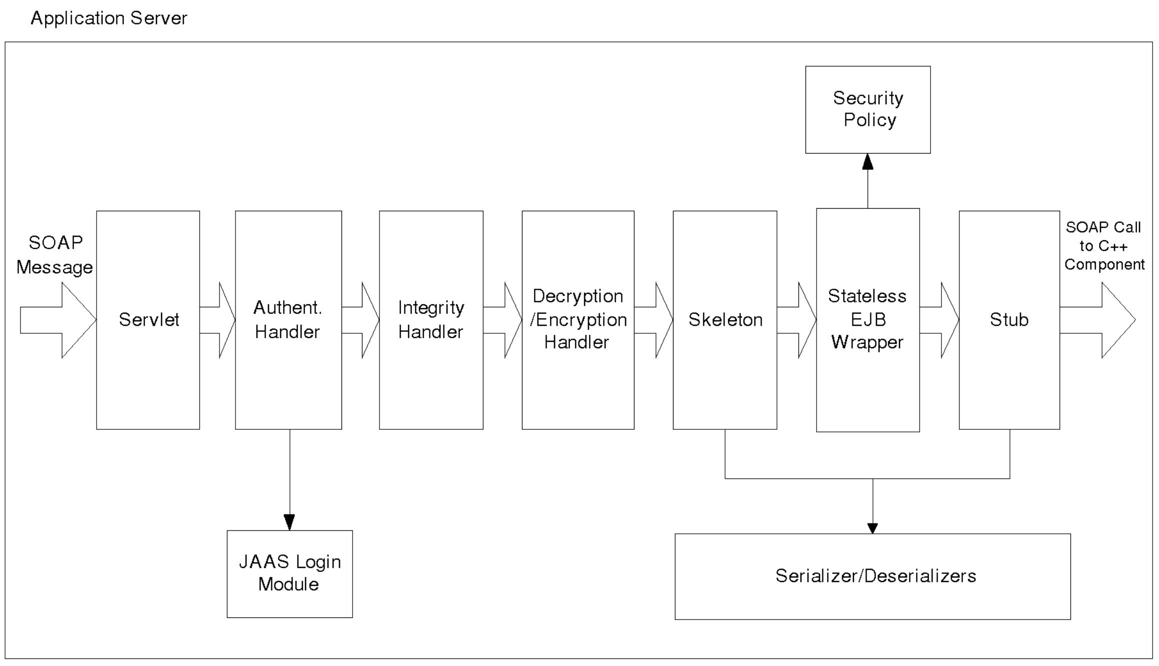


Figure 4. 7 Security in the Reference Architecture

It can be observed that security is implemented with the chain of responsibility design pattern, which effectively decouples the caller from its target by interposing a chain of objects between then [42]. This concept manifests itself in the proposed reference architecture as three handler objects residing between the servlet and skeleton. The first handler performs authentication by essentially pulling the client credentials out of the SOAP header and passing them to the JAAS login module for verification. Upon passing this phase, a security context is created and propagated along the chain to the integrity handler, which then verifies any provided XML digital signatures. The message is then finally decrypted prior to invocation of the EJB proxy object. Based upon the generated context, the EJB container then takes responsibility for ensuring that the client does indeed have access to the desired resource by inspecting the security policy. It is evident that in general, these handlers perform the reverse operations on the service response as it flows through the chain; the authentication handler will, however, only append the server's credentials when mutual authentication is required. It can be noted that in the case that the legacy system is exposed to an interactive thin client, similar principles would apply with the exception that handlers would appear before the servlet in order to decrypt the incoming SSL-encoded data stream.

The preceding figure depicts the enterprise bean invoking the back end service in the clear, that is, free of security mechanisms. This scenario is indeed appropriate when the service is contained within the safe confines of the enterprise firewall. However, in the case that the intranet is not deemed secure or the wrapper is invoking a service outside the trusted domain, the invocation would be required to flow through a similar gauntlet of security functionality.

4.3.2.4 Security and the EWE Toolkit

As mentioned previously, since it is indeed possible to separate the business logic from the security mechanisms, the appropriate handlers and JAAS components can be automatically generated at deployment time. It is evident that different service installations may undoubtedly have disparate security requirements, thus necessitating a highly flexible configuration paradigm. To illustrate this point, the following figure represents a sample config file that developers would be required to produce for the banking application. It can be noted that this file, along with the remainder of the configuration information, would be wrapped with a simple GUI interface in order to mask the underlying XML details from the developer.

```
<security-spec>
    <security-role>
        <description>free access to all banking-related methods </description>
        <role-name>Teller</role-name>
    </security-role>
    <security-role>
        <description>clients using ATM machines not belonging to the bank </description>
        <role-name>ForeignATMClient</role-name>
    </security-role>

    <identity type=”password”>
        <username>Kostas</username>
        <password>Crete</password>
        <role>Teller</role>
    </identity>
    <identity type=”X.509”>
        <name>Boris Becker</name>
        <role>ForeignATMClient</role>
    </identity>

    <method-permission>
        <role-name>Teller</role-name>
        <method>
            <component-name>Banking</component-name>
            <method-name>*</method-name>
        </method>
    </method-permission>
    <method-permission>
        <role-name>ForeignATMClient</role-name>
        <method>
            <component-name>Banking</component-name>
            <method-name>withdraw</method-name>
        </method>
    </method-permission>

    <!-- <transport-security type="ssl"> -->

    <message-security>
        <signature>
            <method>
                <component-name>Banking</component-name>
                <method-name>*</method-name>
                <canonicalization-method>
                    http://www.w3.org/2001/10/xml-exc-c14n#
                </canonicalization-method >
                <signature-algorithm>
                    <symmetric-credential>
                        http://www.w3.org/2000/09/xmldsig#hmac-sha1
                    </symmetric-credential>
                    <asymmetric-credential>
                        http://www.w3.org/2000/09/xmldsig#rsa-sha1
                    </asymmetric-credential>
                </signature-algorithm>
                <digest-algorithm>
                    http://www.w3.org/2000/09/xmldsig#sha1
                </digest-algorithm>
            </method>
        </signature>
        <encryption>
            <method>
                <component-name>Banking</component-name>
                <method-name>deposit</method-name>
                <method-param-name>m_objAccount</method-param-name>
                <algorithm>
                    <symmetric-credential >
                        http://www.w3.org/2001/04/xmlenc#3des-cbc
                    <symmetric-credential >
                    <asymmetric-credential generateKey="true">
                            <X509Certificate encoding="Base64">nSDFSiur…</X509Certificate>
                            <algorithm>http://www.w3.org/2001/04/xmlenc#rsa-1_5</algorithm>
                    <asymmetric-credential >
                </algorithm>
            </method>
        </encryption>
    </message-security>
</security-spec>
```

Figure 4. 8 EWE Security Deployment Descriptor

The <security-role> tags effectively define two roles for the banking application; as the names imply, the 'Teller' is meant to represent a bank teller capable of fulfilling all standard banking applications while the 'ForeignATMClient' represents a client accessing his account from an ATM machine under the jurisdiction of a different bank. The <identity> tags not only map specific users to these roles, but also indicate the type of authentication mechanism required. Based on current industry standards, the toolkit will support four possible authentication techniques: username/password pair, Kerberos tickets, X.509 certificates and SAML assertions. It can be noted that it is only in the password case that the identity (username) is inserted directly into the SOAP message; the other techniques require the JAAS login module to perform decryption in order to uncover the identity. As observed above, the banking application permits a teller named 'Kostas' to authenticate with a password as well as a foreign client named Boris Becker to authenticate with an X.509 certificate. In the latter case, the authentication handler will delegate to the login module, which will in turn crack open the certificate, ensure that the identity exists, and create a security context. The <method-permission> tags complete the definition of the security policy and are used to generate authorization checks in the beans. It is evident that the Teller role has access to all the functions in the banking app while identities in the ForeignATMClient role can only withdraw funds.

In addition to supporting a variety of authentication credentials, the toolkit must also provide flexibility in the selection of integrity and confidentiality mechanisms. As SSL may indeed be sufficient in certain scenarios, the <transport-security> tag is available to enable this option. In this case, however, since the banking application does not actually require SSL, the element is commented out. It is the <message-security> construct that determines the more robust security mechanisms provided by the WS-Security standard. The <signature> element signifies the requirement of an XML signature, while the <method> tag defines the portion of the SOAP message to be signed; though the granularity could be defined to the level of individual parameters, it is specified that the withdraw and deposit functions in their entirety should be hashed and encrypted. The <canonicalization-method> specifies the technique used to ensure that XML documents are represented in a common format so that hashing can be successfully applied. Furthermore, as the name implies, the <signature-algorithm> tag defines the method used to create the digital signature. As such algorithms require a key, it is assumed that the identity credential provided, whether it be a password, certificate, or Kerberos ticket, is indeed sufficient. Depending upon the type of credential, however, it is clear that algorithms for either symmetric or asymmetric key cryptography will have to be applied.

The <encryption> element, structured very similarly to its <signature> counterpart, defines the portion of the SOAP message to be encrypted on the wire. The only real difference in this case lies in the choice of algorithm upon use of an asymmetric credential. In scenarios where performance is crucial and/or there is a large amount of data transfer, it would be unacceptable to encrypt the entire data stream using asymmetric cryptography. As a result, a mechanism is provided by which a client can generate a session key to ensure confidentiality and encrypt it with the server's public key. This scenario is represented in the configuration file by the 'generateKey' attribute being set to 'true' as well as the inclusion of the <X509Certificate> element. In this case, the client is expected to encrypt the 'Account' parameter of the 'deposit' function with a generated session key using triple-DES before finally encrypting this session key with the server's public key using RSA.

Having completed the security configuration for the banking application, the developer can pass this information, along with the component interface, to the EWE toolkit in order to automatically generate the enterprise bean wrappers, servlet, JAAS infrastructure, and three security handlers. It is evident, however, that in order to ensure successful consumption of the legacy system's web service wrapper, the server's security requirements must be exposed to the client. In this case for example, the ability to invoke both banking methods would require a client authenticated with a Kostas/Crete username/password pair, an hmac-sha1 digital signature of the entire operation, and triple DES encryption of the account object. Exposing these requirements is achieved via the use of WSDL security extensions, such that toolkits employed at the client can automatically generate a matching set of security handlers.

4.3.3 Transactions

4.3.3.1 Introduction

Having provided a robust security framework, our attention can now be directed towards creating support for transactions. Simply put, a transaction ensures that only agreed-upon, consistent, and acceptable state changes are made to a system – regardless of system failure or concurrent access to the system's resources. This task is achieved by defining transactional units of work satisfying the following four fundamental properties: atomicity, consistency, isolation, and durability (ACID). Transactions are absolutely invaluable to business architectures and when used properly, can make mission-critical operations run predictably in an enterprise environment. The implementation of a robust transaction management framework is, however, a truly daunting task best left to application server vendors; developers should, as a result, leverage the functionality provided by existing middleware technologies. In the context of this paper, transaction management is relevant in two key ways: the simplistic addition of transaction functionality to existing legacy systems and, as we will discuss in the ensuing chapter, the creation of transactions spanning multiple integrated systems. Accordingly, we will first examine the underlying technologies involved and then apply them to integrate seamless transaction support into our reference architecture.

4.3.3.2 Background

Declarative Transactions

Both EJB and COM+ platforms provide a huge amount of value to application developers by effectively shielding them from the complexities of the underlying transaction manager. Similar to the generation of security constructs, the middleware container automatically guarantees that components are appropriately enlisted in the desired transactions based upon a configuration file. Ensuring that the services are completely unaware of the transactions encompassing them not only drastically simplifies application development, but also mitigates the possibility of error, whether it is within the application or on the client side. It can be noted that based on industry demand, both platforms currently only support a flat transaction model, where a fixed series of operations are performed atomically as a single unit of work.

This robust transaction service is also very flexible, for attributes controlling specific component behaviour within a transaction can be defined at the method level. Specification of the ideal settings for each individual component allows developers to declaratively construct complex yet precise interactions. There are four possible attribute values: 'NotSupported', 'Required', 'RequiresNew', and 'Supports'. As the name implies, a value of NotSupported infers absolutely no transaction involvement; if the invoking method is under the hood of a transaction, it is suspended until the NotSupported method completes. The Required attribute is the most flexible mechanism, ensuring that a component always runs in a transaction; existing transactions are joined while the container begins new ones when necessary. Methods marked with Supports only run in a transaction if the client had one running already while the RequiresNew attribute indicates that a new transaction must always be started. The latter is similar to the behaviour of NotSupported for if a transaction is underway when the component is called, it is suspended during method invocation. It should be noted that at the time of this writing, these attributes could only be specified at the component level in COM+; however, future releases will indeed support a lower level of granularity.

Transactions and Web Services

In traditional computing environments, most transactions are executed within the scope of the enterprise, within one trust domain, and with all the resources under the control of one transaction manager. However, the vision of a world of co-operating, distributed web services has introduced the need for more flexible functionality than that provided by existing transaction manager implementations. Firstly, as transactions will be required to span applications implemented with different technologies on different platforms that store state in heterogeneous databases, transaction context propagation must be standardized. Both MS and java platforms currently specify that transaction context should be propagated over the wire in distributed scenarios with the use of proprietary, incompatible protocols (DCOM and IIOP respectively). As a result, interoperability is currently so poor that even standards compliant application servers written by different vendors are not even guaranteed to successfully participate in a distributed two-phase commit transaction.

Furthermore, classic ACIDic transactions may not be suitable for loosely coupled environments such as web services. In this type of scenario, transactions can be very complex, involve many parties, and can potentially last for hours or even days. For example, transactions between a manufacturer and its suppliers might only be considered completed once all parts are delivered to their final destination, which could be days or even weeks after an order is placed. In order to prevent both the reduction of transaction concurrency to unacceptable levels as well as the possibility of denial of service attacks, participants will not want to lock their resources for such extended periods; it is therefore evident that the isolation property must be relaxed in such transactions. Moreover, there are an abundance of scenarios where optional subtransactions are necessary, thus also requiring that atomicity be relaxed. This need is exemplified by a travel agency attempting to reserve seats on a number of airlines at a consumer's request; the global transaction should be able to commit despite one or more of the reservations rolling back due to space limitations or budget constraints. As it is clear that a new model is required to meet these needs, a 'business transaction' has been defined as a consistent state change in the business relationship among two or more parties with each party maintaining an independent application system that maintains the state of each application .

The Future

Two predominant standards have emerged to address these two new requirements: WS-Transactions, which is dependent on existing web service specifications and the Business Transaction Protocol, which defines an abstract XML message set and a SOAP/HTTP binding [50]. Though it is still unclear at this point which spec will emerge victorious, the current web services revolution virtually guarantees that both Microsoft and J2EE platforms will soon be extended to embrace this new XML-based, distributed model. Existing transaction managers will, in all likelihood, be extended incrementally; support for fully interoperable, XML-based 2PC ACID transactions will first be rolled out followed soon afterwards by an implementation of the more flexible business transaction model.

4.3.3.3 Reference Architecture

It is evident that as the vast majority of multi-client, enterprise level applications require transactional support, it is imperative that developers have the capability to easily transaction-enable their applications. However, as mentioned previously, the implementation of the required transaction management functionality is not only out of scope, but also far too complex for most developers, thus requiring that existing middleware technologies be leveraged. Figure 4.9 below depicts how this task can be accomplished by extending the proposed reference architecture to include several transactional components.

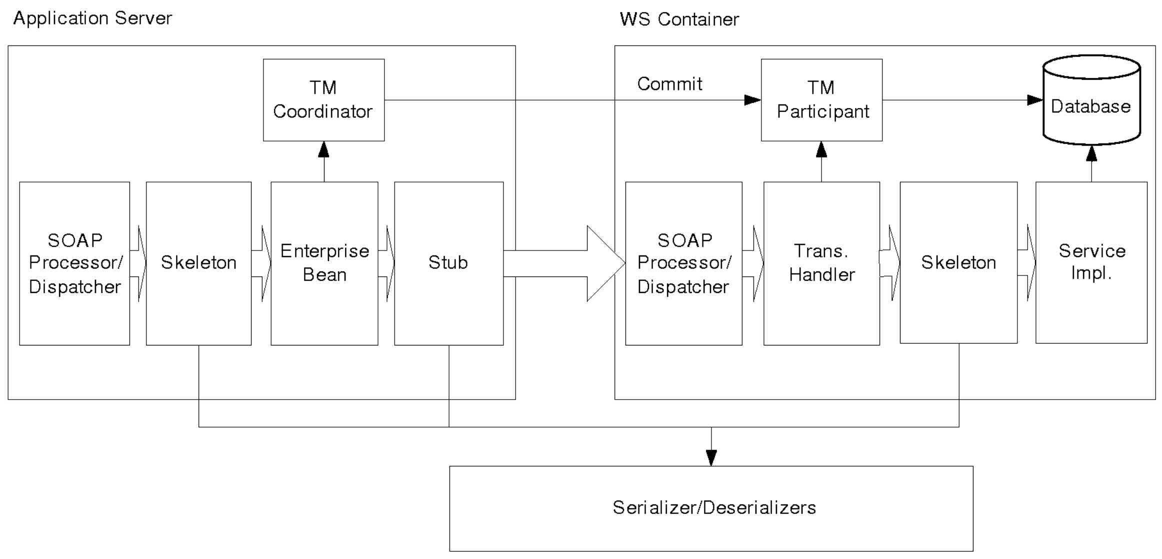


Figure 4. 9 Transaction Management in the Reference Architecture

As we are attempting to merely provide intra-application transactional support, it is evident that only the model for XML-based ACIDic transactions needs to be leveraged at this point; the need for more complex business transactions will become apparent when integration is discussed in chapter five. As observed above, the EJB container interacts with the application server's XML-based transaction manager to demarcate transaction boundaries as required. It can be noted, however, that in order to interact with back end resource manager(s), an additional component, depicted above as the transaction manager (TM) participant, is required. This module is actually provided by application server vendors and can be easily installed and configured with the desired persistent resources. The chain of responsibility design pattern has once again been applied in this case to ensure that a distinction is made between application and transaction logic.

Using the banking application as a simplistic example, let us assume that the withdraw() and deposit() methods are to be placed under the hood of a transaction. Upon invocation of the withdraw() method within the banking EJB, the container interacts with the coordinator to first begin the transaction and then ensure that the transaction context is included in the header of the SOAP message destined for the back end service. At the service installation, the handler processes the appropriate header and invokes the TM participant, which in turn communicates the necessity to begin a transaction to the underlying database resource manager. Once similar steps have been carried out to call the deposit() method, the EJB container signals the coordinator that the transaction must be committed. In general, a two phase commit protocol would be carried out to accomplish this task; however, as there is only one participant in this case, a single commit message can simply be issued in order to improve performance. It should be noted that as this type of distributed transaction mechanism does indeed incur some overhead, simple applications such as the one cited above may be better served making SQL commit and rollback calls directly to the database management system (DBMS). However, more sophisticated applications that are destined for integration, that interact with multiple transactional resources, or that have complex transactional workflows would undoubtedly be well served by this type of architecture.

4.3.3.4 Toolkit Extensions

Since transaction logic need not be embedded within the application code, the existing middleware model for declarative transactions should be adopted by the EWE toolkit. A configuration file, exemplified below in Figure 4.10, can be composed specifying one of the four possible transaction attributes (UnSupported, Supported, Required, RequiresNew) for each component and/or method. Based upon this information, a middleware deployment descriptor can be generated such that the container will automatically control the transactions at the application server. Furthermore, the transaction handler can be generated and included in the legacy deployment to interact appropriately with the TM participant. It should be noted, however, that although the vast majority of transaction management can be handled by the platform, the application developer must specify the error conditions upon which transactions must abort.

```
</transaction-spec>
    <method>
        <component-name>Banking</component-name>
        <method-name>withdraw</method-name>
        <transaction-attribute>Required</transaction-attribute>
    </method>
    <method>
        <component-name>Banking</component-name>
        <method-name>deposit</method-name>
        <transaction-attribute>Required</transaction-attribute>
    </method>
</transaction-spec>
```

Figure 4. 10 EWE Transaction Management Deployment Descriptor

Figure 4.10 provides a rather self-explanatory configuration file for the simplistic banking application. In this scenario, both methods are marked with a 'Required' attribute, indicating that they will always either join or begin a new transaction if one does not already exist. However, if, for example, the deposit method is only required to be part of a transaction when preceded by a withdrawal during a bank transfer operation, it should be marked with the Supports attribute. This setting would ensure that it could be involved in other workflows where transactions are not required.

Unlike the security requirements outlined in the previous section, as the client is, in general, unaffected by the transactional nature of the system, the method by which the legacy application is exposed is completely independent of the transaction management functionality. However, when exposed as a web service, knowledge of a system's transactional behaviour provides developers with a more complete view of the installation, thus providing them with all the knowledge necessary to compose successful clients. Accordingly, the transactional attributes for each method should also be included in the WSDL description.

4.3.4 Clustering

4.3.4.1 Introduction

The generation of a secure, transactional environment is not sufficient for truly robust, mission critical applications; clustering capability is also required in order to ensure that systems are both highly available and scalable. Organizations seeking to web enable their critical legacy assets will undoubtedly require that these qualities be inherent to their newly developed web platforms. A cluster can be defined, quite simply, as a loosely coupled group of servers that provide a unified, simple view of the services that they offer individually. This concept provides availability by effectively minimizing the single points of failure in a system and ensuring seamless fail-over in the case of hardware or software failures. Furthermore, scalability is not only guaranteed by the implementation of a load balancing algorithm, but also by the ability to increase capacity with the addition of supplementary servers. Similarly to security and transaction frameworks, as the realization of an effective, maintainable cluster is a complex task, it should be left to the application server. To this end, we will examine the clustering capabilities of existing middleware technologies and propose a reference architecture by which web enabled legacy applications can easily leverage this functionality.

4.3.4.2 Background

Application Server Clustering

Existing component middleware technologies have extensive clustering support built directly into the platform. Firstly, deployment and maintenance across the cluster is drastically simplified for the cluster can automatically synchronize itself once files have been pushed out to a single machine. It can be noted that the seamless propagation of file updates or configuration changes throughout the installation with minimal, if any, downtime is an absolutely critical feature, especially as the cluster size increases. Many vendors also package dynamic application launchers, which have the ability to automatically restart applications in case of failure. Furthermore, load balancing and failover logic is automatically generated and generally placed in the remote stubs of the middleware components. In essence, a stub is aware of several identical remote objects available for use in the cluster and will select one as appropriate upon each invocation. It can be noted that fail-over occurs at two levels. In the case of an idle machine crash, the appropriate stubs will detect this state change and cease to invoke the culprit. Moreover, if a machine fails during invocation, fail-over to a backup is still indeed possible given that the method is idempotent. Even in the case that client state is maintained at the server, it is periodically replicated (generally on transaction boundaries) to a redundant machine, thus allowing for seamless session fail-over.

4.3.4.3 Reference Architecture

Unlike the realization of the security and transactional frameworks, there are a host of possible clustering architectures depending upon the nature of the application as well as the specific system requirements. Based upon the type of content delivered (static vs dynamic, presence of session data), the tolerance for single points of failure, the security requirements, as well as the budget restrictions, the clustering technologies and methodologies employed can vary drastically. The fundamental point in this case is that since the creation of a serviceable, robust clustered environment is truly infeasible for developers undertaking web enablement projects, the clustering logic must be pushed into the middleware wrapper. This objective can be achieved with two main strategies: the three-tier architecture pushes the web and middleware content onto the same machine while four-tier architecture distributes them to separate hosts.

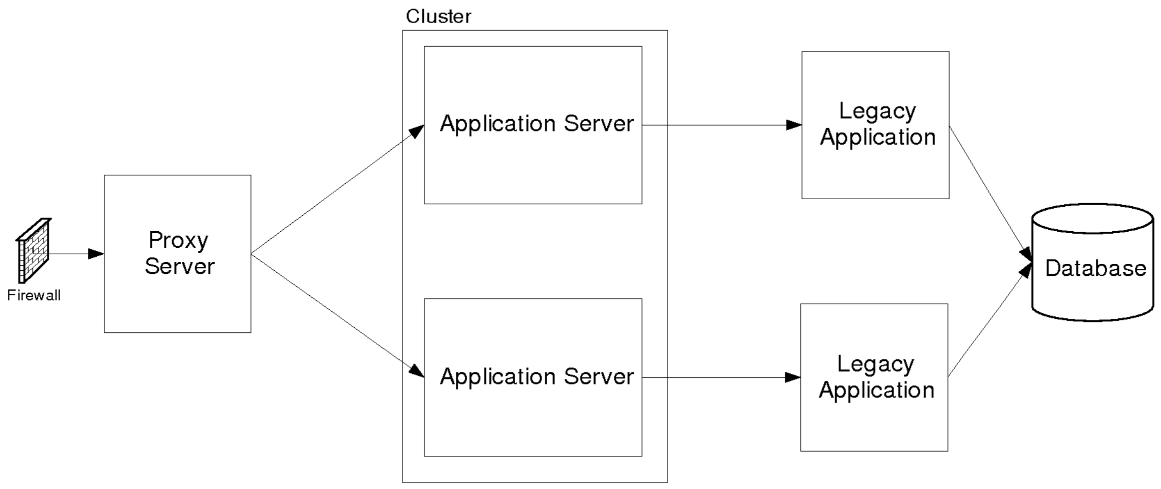


Figure 4. 11 4-tier Clustering in the Reference Architecture

The preceding figure represents a possible implementation of the latter scenario; in this case, a server, containing merely the web content (static and dynamic), acts as a proxy for the application servers hosting the middleware components. The middleware component stubs, which reside on the proxy server, are relied upon for both load balancing and fail-over logic. Furthermore, the legacy application is installed independently on separate machines at the back end with each instance mapping one to one with an application server. It is evident that what this architecture gains in simplicity, it loses in flexibility; pairing the legacy tier and the middle tier in this fashion could well squander capacity unnecessarily. For example, let us assume that the application servers can handle more load than the back end machines. If one of them were to fail, requests would no longer be directed to its properly functioning legacy mate, thus wasting precious server resources. Finally, in the scenario depicted above, the proxy server acts as a single point of failure; however, a redundant machine could be added without much difficulty.

A three-tier architecture with 'smart middleware' is portrayed below in Figure 4.12. Since the web and middleware tiers reside on the same machine in this case, a hardware unit is used to manage load balancing and fail-over. It should be noted, however, that the application servers will still handle fail-over in the event of a machine crash following method invocation. The middleware servers are considered to be 'smart' in this scenario, for they are responsible for providing both load balancing and fail-over at the back end tier. Unlike the clustering functionality provided in the middleware stubs, however, this logic is not inherent to the platform and must therefore be generated as application logic within the components by the EWE toolkit.

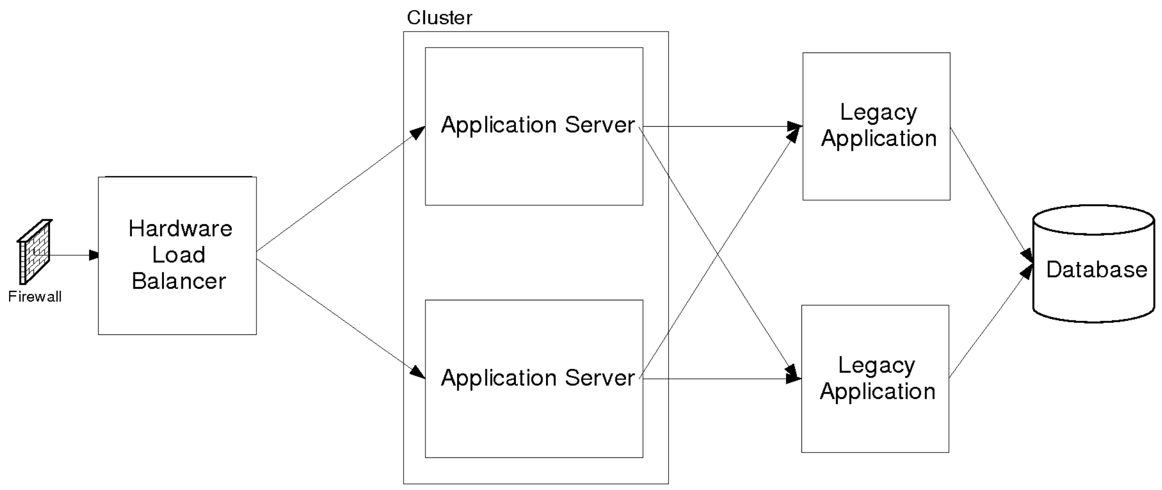


Figure 4. 12 3-tier Clustering in the Reference Architecture

4.3.4.4 Toolkit Extensions

In contrast to the generation of transaction and security mechanisms, it is evident that the creation of clustered environments similar to those outlined above cannot be fully automated. Depending upon the desired architecture, there are a number of steps that the service provider must execute. Firstly, the back end application must be manually installed on several different machines; though this process may appear to pose a maintenance nightmare, it is generally assumed that legacy applications will not be altered. Having completed this task, the EWE toolkit can then generate the appropriate web components and middleware wrapper based upon a configuration file, exemplified below in Figure 4.13. Though this file is sufficient for the automatic deployment of the middleware throughout the cluster, additional steps must be taken in the event that a three-tier architecture is employed. In this case, an additional load balancing mechanism, whether it is a DNS round robin or a hardware device, must be configured in front of the cluster.

```
<cluster-spec>
    <architecture tier=”3”>
        <smart-middleware>
            <algorithms>
                <load-balancing>
                    <round-robin />
                </load-balancing>
                <fail-over>
                    <ping interval=”30” />
                </fail-over>
            </algorithms>
        </smart-middleware>
    </architecture>

    <component>
        <name>Banking</name>
        <locations>
            <location>
                <machine-name>1.2.3.4</machine-name>
                <port>1000</port>
            </location>
            <location>
                <machine-name>1.2.3.5</machine-name>
                <port>2000</port>
            </location>
        </locations>

        <method>
                <name>withdraw<name>
                <idempotent>false</idempotent>
        </method>
        <method>
                <name>deposit<name>
                <idempotent>false</idempotent>
        </method>
    </component>
</cluster-spec>
```

Figure 4. 13 EWE Clustering Deployment Descriptor

The preceding document represents a clustering configuration file for the banking application. As the name implies, the <architecture> tag describes the characteristics of the entire installation; in this case, the web and middleware components will be deployed on the same machine and clustering logic for managing the legacy tier will be generated within the middleware layer as denoted by the <smart-middleware> tag. It can be noted that in the case that a four-tier architecture is specified, the enterprise toolkit would generate separate deployment descriptors for the web and middleware layers. The file indicates that logic will be generated in the middleware to not only implement a simplistic round robin load balancing algorithm, but to also ping the legacy hosts every thirty seconds to ensure that they are still alive. The <component> element then encapsulates the relevant information for each of the exposed legacy modules. It is evident that the banking component is available on two separate hosts and both of its constituent methods are not idempotent, thus indicating that fail-over is not possible once they have been invoked.

4.4 Conclusion

Countless corporations worldwide are struggling with the dilemma of salvaging the enormous sunk cost poured into their now archaic information systems; many have invested millions in new installations while others have spent countless amounts on fruitless integration efforts. In this chapter, we offer a simply solution to this problem, whereby existing legacy assets are web enabled, thus promoting their reuse both within and outside the enterprise. We have proposed a reference architecture, which effectively involves the wrapping of legacy systems with both middleware and web components using XML web services infrastructure as the underlying communication mechanism. In addition to exposing the back end application to both thin clients browsers and business partners via SOAP, as we've outlined in great detail, the wrappers also serve to add robust security, transaction, and clustering functionality to the installation. Moreover, despite the performance concerns, the use of XML as the technology bridge to the legacy system creates a very flexible, data-centric solution; the integration philosophy is essentially shifted from a world of proprietary interfaces to one of open, standardized data transfer. Furthermore and perhaps most importantly, we have proposed the idea of the enterprise web enablement (EWE) toolkit, which automatically generates all the components necessary to realize the proposed architecture. In essence, based only on interface descriptions and a variety of configuration information, a legacy system can be automatically transformed into a fully-featured, enterprise level web app. In the ensuing chapter, we will take this solution to the next level by leverage the EWE toolkit to not only web enable, but also integrate a multitude of disparate systems within the enterprise.

Chapter 5
Fashioning the Roadmap toward the Integrated Enterprise

5.1 Introduction

As outlined in the first chapter, years of development, technological innovations, mergers and acquisitions have left many corporations in possession of a multitude of disparate, incompatible information systems. There is no doubt that the ever increasing customer and business partner expectation for real-time information has forced companies to attempt to link these systems in order to improve productivity, efficiency and, ultimately, customer satisfaction. However, increasing competition and shrinking budgets have left managers scouring for innovative, cost-effective methods to achieve this elusive task. Integration efforts to date, focused predominantly on the development of proprietary point-to-point adapters, have proven to be a daunting task with countless failed projects and losses in the millions. As a result, in addition to the web enablement of various legacy assets, there is an overwhelming need for a simplistic integration methodology, which will yield a common homogeneous framework. It was noted in the previous chapter that the proliferation of XML web services into the enterprise has the potential to revolutionize existing integration strategies. The cost savings and ease of implementation associated with wrapping virtually all legacy systems, past, present and future, with standardized, code-independent, data-centric interfaces is truly astounding. Though application SOAP-enablement does indeed make for simple invocation from virtually all remote clients, it is, however, only part of a successful integration solution. In order to amalgamate a variety of heterogeneous applications, a network-aware integration platform must also be present. As we will explore in the ensuing sections, this type of environment maps directly to the reference architecture outlined above and can be seamlessly created with the Enterprise Web Enablement toolkit.

5.2 Reference Architecture for Integration

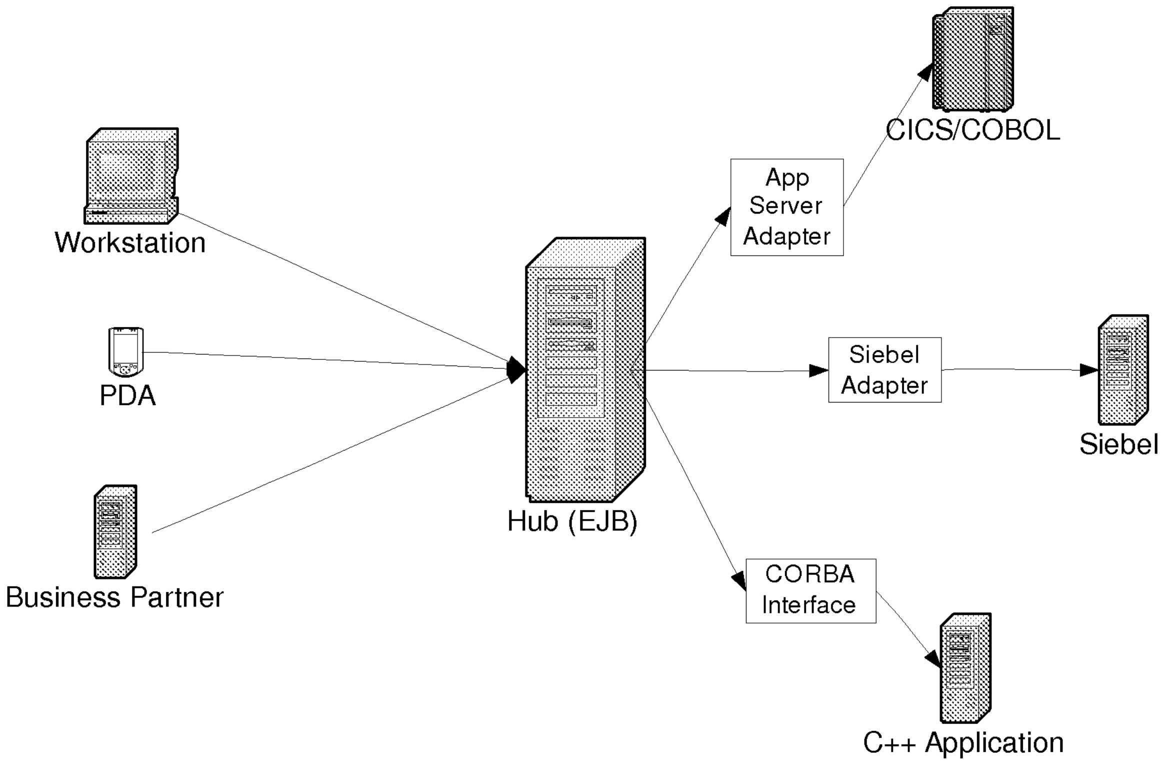
In order to portray precisely how the proposed reference architecture and EWE toolkit can be applied to the integration of a multitude of differing systems, a simplistic business case involving a phone company will be traced throughout the remainder of this section. Let us assume that over the years, the corporation has garnered a legacy CICS/COBOL call center application running on a mainframe, a Siebel Customer Relationship Management (CRM) package, a custom built financial application written in C++, as well as a web-based J2EE application to manage their internet subscribers. Written in the mid-eighties, the incredibly complex call center system was designed to manage all of the information surrounding a customer's phone service. As the company continued to grow, however, the user interface became a limitation, thus leading to an expensive, proprietary migration to the client-server paradigm. The eventual completion of this task yielded a fat client installed on each call center machine across the country communicating with the same centralized, mainframe server. Expanding revenue streams then led to further demand for increased IT functionality; the need for tools to manage product purchases, inventory, invoicing, preventive maintenance, marketing, as well as to manage business partner relations began to emerge. The third party CRM product was therefore purchased to address these new requirements; it was decided, however, that the cost and risk associated with the replacement of the existing call center application was far too great, thus leaving the enterprise with two separate views of their customer base. Furthermore, the financial application, designed to address specific billing and accounting needs, processes batch files, which effectively log the relevant transactions over the course of a business day. Finally, the J2EE application was built as a completely separate entity to manage the exponentially increasing number of consumer and business internet accounts.

It is evident that decades of development yielded a variety of incompatible legacy assets, thus leaving the corporation with a Pandora's Box of pressing problems. First and foremost, the existing infrastructure has produced a fragmented view of the customer; the call center system holds information pertaining to the actual telephone service, in addition to the retail sale data, the CRM package manages the majority of the business intelligence, the financial application hosts the necessary accounting functionality, and the self-contained J2EE application contains the customer data pertaining to internet service. This separation is both costly and error prone, for data must be manually updated and maintained across the enterprise; the creation of a new account in the call center system, for example, requires that the information be tediously keyed into the other three applications. In addition to automating this task, there is no doubt that a homogeneous, real-time IT platform would also allow the corporation to provide a far superior, dynamic customer service experience.

Furthermore, as the completion of every new release of the call center application requires re-installation to thousands of client machines, an annual price tag reaching well into the millions is incurred. It can be noted that this overhead can be virtually eliminated by migrating to a thin client browser interface; in addition to mitigating call center client upgrades, functionality could also be pushed directly to the customer (e.g. phone line disconnection). Logically speaking, perhaps the most apparent solution to these problems would involve a complete re-write of all the systems to a common web-driven platform; however, as outlined in the second chapter, leveraging existing assets is a much more viable avenue. It is therefore clear that the systems must be amalgamated within a common, web-enabled framework.

5.2.1 Hub and Spoke Architecture

A number of standardized network topologies have been implemented to address a wide variety of integration issues; the most widely accepted solution, however, is the simplistic hub and spoke architecture. As seen below in Figure 5.1, a centralized server acts as a façade for a host of legacy systems, thus essentially serving as an integration platform for the entire enterprise. It is clear that once the back end applications have been successfully integrated with the hub, they can then seamlessly communicate with one another and/or have their data pushed to a variety of different client channels. Furthermore, in addition to addressing the underlying plumbing requirements of any robust server installation (threading, connection pooling, instance pooling, load balancing, fail-over etc.), it can be noted that middleware technologies are generally employed at the hub to provide a common security and transactional framework for the entire information system.



5. 1 Generic Hub and Spoke Architecture

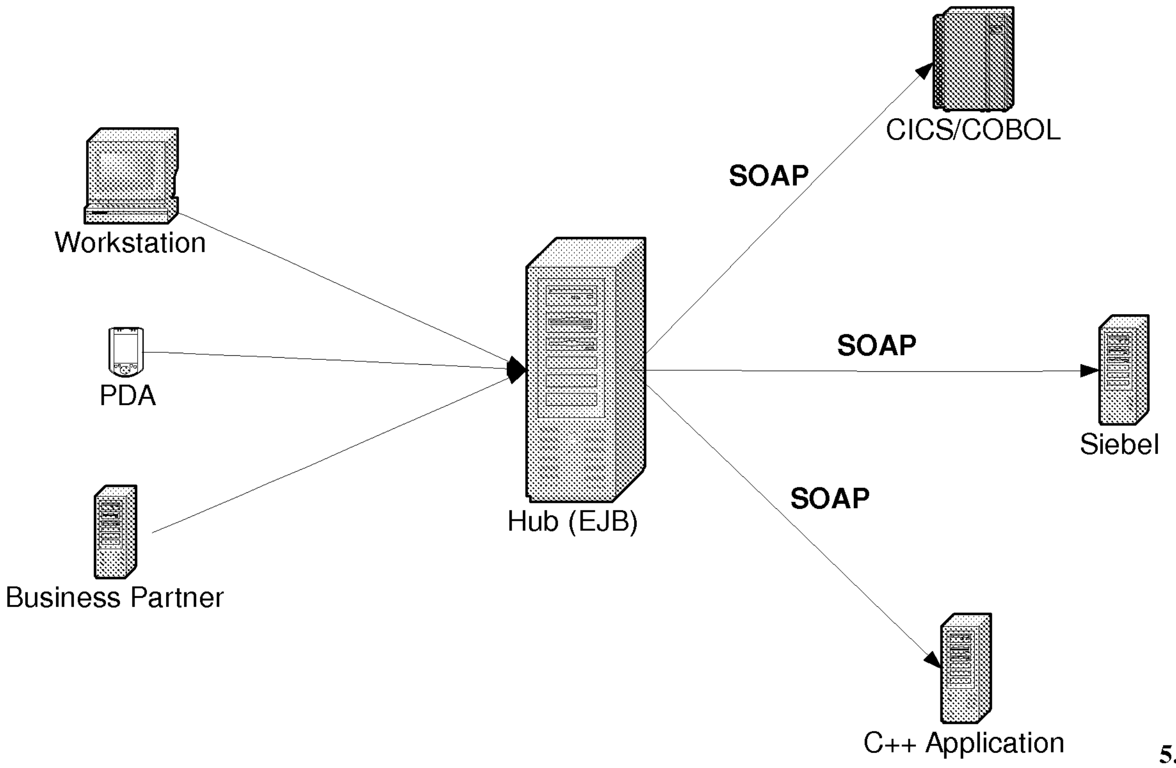
The preceding diagram depicts the implementation of this hub and spoke architecture at our hypothetical telephone corporation. It can be observed that having already elected the java platform in one of their previous undertakings, the logical decision is the selection of EJB technology at the hub. The back end applications are then integrated in the most appropriate manner; the gaps to the CICS/COBOL and CRM systems are bridged with proprietary adapters provided by the application server vendor and Siebel respectively, the C++ application is exposed as a set of CORBA components, and as it is already composed of JSP pages and EJBs, the internet management application is deployed directly into the hub itself. In addition to this integration exercise, developers also have to create a thin client front end for the relevant applications either from scratch or via the use of a migration tool.

Upon the completion of this integration process, it is evident that the resulting system successfully solves virtually all of the aforementioned problems. Firstly, it can be noted that by simply adding functionality to the hub, the legacy applications can be extended without the risk and complication of modifying the original code. This feature is certainly most beneficial in the case of the call center application, where the code is both complex and tightly coupled and the initial developers more than likely left the company years earlier. Furthermore, as mentioned previously, the web enablement of the call center application not only clearly solves the deployment problem, but even allows certain portions of the front end to be accessed by customers directly. The most compelling benefit of this methodology, however, is the creation of an enterprise wide API. As the hub essentially contains EJB wrappers for the functionality of all four systems, custom applications or third party workflow solutions can be composed to access everything from customer phone service data to marketing information in real-time. A workflow for account creation through the call center could, for example, be easily created; after the information is keyed into the system by a representative, the system would automatically check for an account in the CRM system. If no account exists, one is immediately created; otherwise the CRM package updates its records with the pertinent information. The order is then finally passed along to the financial application for accounting purposes. In addition to reducing both errors and maintenance costs, this solution also drastically improves efficiency. From providing a unified front for both phone and internet services to providing the capability to email relevant information to customers immediately upon sign up to the instant update of online information after speaking to a representative, this architecture clearly provides a huge amount of added value to the customer.

Despite these compelling benefits, this architecture does, however, have one major drawback. Firstly, it can be noted that although the hub could be perceived as a bottleneck for the system, the processing power currently available in addition to the clustering capability of the middleware are indeed sufficient to handle virtually any load. As outlined in the previous chapter, there are a number of feasible distributed solutions; in this case, however, to enforce a form of modularity, the system could be deployed and load balanced in such a manner that the internet management application would reside on its own separate machine. The more relevant shortcoming of the installation, however, is its error prone, brittle nature resulting directly from the selected integration strategies. Proprietary adapters essentially perform the appropriate mappings amongst specific technologies; J2EE application server vendors will, for example, compose adapters to connect a host of different platforms, legacy and otherwise, to their EJB environment. In many cases, however, these off the shelf adapters need sophisticated configuration and tuning or are not available at all, thus leaving the task to highly trained consultants. It is evident that in addition to being extraordinarily costly, this methodology produces very tightly coupled solutions, which will undoubtedly require further integration effort. Any minor change to the legacy system will entail a modification, compile, test cycle of the adapter while complete replacements of either the hub technology or legacy app will clearly necessitate a full re-design of the integration strategy. It can be noted that exposing the financial application via a set of CORBA interfaces is indeed a finer approach, for it is standards-based solution that mandates the use of encapsulation. As mentioned previously, CORBA is, however, an older, complex middleware technology that is lacking in industry momentum and requires a significant amount of training for successful implementation. We firmly believe that the time, cost, effort, and risk associated with the development of two distinct proprietary adapters as well as a CORBA implementation far outweigh the tremendous benefits of the hub and spoke architecture; it is, as a result, apparent that another integration solution must be crafted.

5.2.2 Applying Web Services and the EWE Toolkit

As outlined in the previous chapter, leveraging XML web services for integration essentially shifts the focus from proprietary mappings and interfaces to standardized data transfer. In the context of the hub and spoke architecture, as depicted below in Figure 5.2, a huge amount of flexibility is gained for the constituent components are simply required to produce and consume standardized XML messages. As a result, upon the modification or replacement of the hub or legacy applications, it must simply be ensured that the resulting systems have the ability to process the same set of data messages. Furthermore, as alluded to earlier, XML messaging can be applied to virtually any system, thus precluding the need to adopt multiple integration enablers. These undeniable benefits coupled with the open, simplistic nature of web services as well as their dependence on ubiquitous, established technologies have given them tremendous industry traction, essentially cementing the technology firmly into the future of enterprise computing. As a result, we strongly believe that the time is rapidly approaching that systems will be designed from the ground up or retrofitted for integration using SOAP. To this end, it can be noted that although intra-enterprise integration has been the sole topic of discussion to date, web services' dependence on HTTP beckons the notion of inter enterprise communication. The hub could therefore consolidate a host of systems extending from within the corporate intranet across the internet; one could envision, for example, our phone company integrating with a hosted third party ERP solution or an independent credit validation service.



5. 2

5.2 SOAP-enabled Hub and Spoke Architecture

These tremendous benefits notwithstanding, the most compelling advantage of this XML messaging concept is the ease of implementation. It can be noted that the model outlined above is merely a different representation of the reference architecture proposed in the previous chapter. As a result, having selected the desired hub technology, in order to automatically generate the underlying XML communication infrastructure as well as the middleware and necessary web components, a developer must simply apply an EWE toolkit specific to each legacy application. Continuing our example, upon identification of the desired function points, COBOL/EJB and C++/EJB toolkits are be applied to the call center and financial applications respectively; the generated wrappers are each deployed to a central location. It can be noted that as the source code for the CRM package is unavailable, the onus is therefore on the vendor to provide the tooling with which to carry out the necessary integration tasks. Furthermore, based on the appropriate configuration information, the middleware is generated in such a way that the security and transactional requirements of each of the back end systems are accurately captured. In fact, the standards based nature of transaction context propagation would even permit transactions to easily span across multiple legacy applications. Moreover, it is interesting to note that the shift towards XML-based standards compliance in the world of modelling business processes (e.g. BPEL) suggests that the system is not strictly confined by the middleware technology selection [51]. Let us assume, for example, that following the completion of a BPEL workflow management system, the phone company wished to port their entire integration environment to the Microsoft platform. In this case, it would simply be a matter of applying a different set of tools; the middleware would first be regenerated with a MS-based EWE toolkits and the relevant BPEL document could then be recompiled in this new environment.

5.3 Conclusion

The example portrayed throughout this chapter is indeed indicative of the overwhelming need for integration in industry today. Though there are a host of possible solutions, the hub and spoke architecture represents a simple, generic methodology to solve this global problem. The creation of a centralized, network-aware, enterprise-wide installation is indeed a powerful concept that serves to not only drastically cut down deployment and maintenance costs, but will also ultimately lead to far superior customer service. Widespread adoption of this architecture has, however, been stagnant due to the fact that traditional implementations have relied on the use of complex, costly, often unviable integration strategies. The arrival of XML web services has indeed provided light at the end of the tunnel; the use of established protocols and standardized message formats coupled with the ease of implementation provide a simple, cost-effective means to connect heterogeneous nodes, both within and outside the enterprise, to the hub. In fact, a SOAP-aware hub and spoke implementation can be created merely by applying the Enterprise Web Enablement toolkit proposed in the previous chapter to the desired legacy systems. The flexibility and extensibility of this installation leaves little doubt that it will represent the strategy of choice for the vast majority of future integration projects. Moreover, web services are indeed here to stay, thus ensuring that the adoption of this approach will be the final significant integration investment for many years to come.

Chapter 6 Conclusion

6.1 Overview and Findings

An industry-wide inability to manage the evolution of the 18 trillion dollar information technology investment of the last decade has left enterprises with a vast array of truly heterogeneous systems and islands of information. Failure to effectively consolidate this data, coupled with a difficulty to meet the changing business and technical requirements of pushing this content outside corporate boundaries, has ultimately driven up costs, degraded efficiency, and ultimately impacted customer satisfaction. Integration attempts to date have, in general, proven to be fruitless with countless failed projects and losses in the millions. The advent of XML web services does, however, provide a potential solution for companies desperate to leverage their IT assets. In addition to possibly creating a wealth of reusable, distributed components across the internet, we feel that the power and the simplicity of the web services platform can and must also be leveraged for intra-enterprise integration.

Before this methodology can come to fruition, however, it is evident that the current interoperability issues of the web services model must be rectified. To this end, we feel that the concepts employed in the WSDL specification are sufficiently flawed to merit the proposal of several key amendments. The excessively complex, verbose WSDL spec mandates the application of an extensive, convoluted, contradictory set of rules for SOAP message creation. As a result, messages generated by one toolkit are often not recognized or even worse, are processed incorrectly by another. In order to achieve interoperability, XML messages must, quite simply, mean the same thing to every endpoint; servers must essentially be afforded the ability to validate incoming requests at the message level. To this end, WSDL documents must therefore provide a schema definition for precisely what must appear on the wire; this simplistic approach removes any ambiguity and introduces instant interoperability into the platform. This paradigm shift can be achieved with several essential changes to the WSDL spec, including mandating the use of XML Schema, the removal of the <wsdl:message> construct, as well as the elimination of both the 'encoded' and 'rpc' binding options.

Having addressed the major interoperability concerns of the web services model, it was then leveraged to define a reference architecture for the web enablement of legacy assets. Similar to existing wrapping methodologies, the use of middleware and web component wrappers is proposed; however, in this case, SOAP, and not a proprietary adapter, is relied upon as the bridging technology. Posing a radical change in distributed computing, XML's complete independence from programming languages precludes the need for legacy applications to fit into a proprietary, program infrastructure before integration can take place. The legacy systems and middleware tier are merely required to produce and consume standardized XML messages, thus giving the architecture a huge amount of flexibility and extensibility by effectively decoupling the two layers. Furthermore, in addition to being far simpler to implement than other techniques, this solution offers the proverbial 'silver bullet', for it can be applied to virtually every existing technology.

In an attempt to ensure the adoption of this architecture, the 'Enterprise Web Enablement' (EWE) toolkit is proposed; based on a legacy interface, it will automatically generate both the middleware and underlying XML communication infrastructure as well as a web tier serving to expose the back end applications to both thin client browsers and business partners via SOAP. As a result, upon the installation of the toolkit and an application server, a developer is able to automatically transform a legacy system into an enterprise-scale web installation. Furthermore, the toolkit can be configured to enhance the existing system by producing secure, transactional, and/or clustered middleware components.

Finally, the scope of this architecture is extended from web enablement to complete enterprise integration. We outline how a hub and spoke architecture is automatically created from the application of EWE toolkits to several legacy systems. This model produces a loosely-coupled homogeneous enterprise, allowing legacy systems to be easily modified, updated, and/or replaced. Moreover, an enterprise-wide API is generated at the hub, thus allowing additional applications or workflow solutions to simultaneously leverage the disparate legacy assets. We strongly believe that the open, simplistic nature of web services, coupled with their dependence on ubiquitous, established technologies will give them tremendous industry traction, thus essentially cementing the technology firmly into the future of enterprise computing. In fact, the time is rapidly approaching that systems will be designed from the ground up or retrofitted for integration using SOAP. As a result, given that the health of legacy systems is preserved, the implementation of the integration strategy proposed in this paper will undoubtedly serve to save corporations millions in integration and redevelopment costs.

6.2 Future Work

There are indeed a number of practical projects that must be carried out to verify the proposals detailed in this body of work. Firstly, the proposed changes to WSDL can be incorporated into web services toolkit implemented for different platforms. In order quantify the increased interoperability, their communication can then be compared with that of existing toolkits. Furthermore, the EJB-C++ EWE toolkit prototype must be extended to completion; an analysis of scalability and deployment issues for larger industrial software components can then be carried out. More interestingly, true validation of the concepts presented here will only occur upon the implementation of a toolkit for a mainframe technology, such as CICS/COBOL.

Bibliography

[1] Sneed, H.M., “The Rationale for Software Wrapping”, in the proceedings of the International Conference on Software Maintenance, Bari, Italy, October 1-3, 1997.

[2] Gould, P., “Integrating Legacy Systems”, http://eai.ebizq.net/bpi/Gould_1.html

[3] Aberdeen Group, “A case study of the EAI marketplace”, http://eai.ebizq.net/leg/collins1.html

[4] Gordon, H., “Unlocking Your Investment in Mainframe Business Processes”, http://www.webmethods.com/PDF/Unlocking_Investments_in_Mainframe_Processes.pdf

[5] Gosin, Sanjay, “EAI: The Business Drivers and Technical Challenges”, University of Maryland, Oct 15, 2001.

[6] Briggs B., “The Real Time Enterprise”, http://www.aptsoft.com/overview.htm

[7] Govekar, Milind, “EAI – The Magic Glue”, http://www.vnunet.com/News/1139489j, Gartner Group, 2001

[8] Fagan, Brendan, “Identifying major integration hurdles”, Forrester Research, 2001.

[9] Brodie, M./Stonebraker M.: Migrating Legacy Systems, Morgan Kaufmann Pub., San Francisco, 1995, p. 41.

[10] Ewusi-Mensah, K., “Critical Issues In Abandoned Information Systems Development Projects”, Comm of ACM, Vol 40, No. 9, Sept. 1997, p. 75.

[11] Sneed, H.M., “A case study in software wrapping”, in the proceedings of the International Conference on Software Maintenance, 16-20 Nov., 1998.

[12] Chikofsky, Elliot and Cross, James. “Reverse Engineering and Design Recovery: A Taxonomy”, IEEE Software, January, 1990.

[13] Terekhov, A./Verhoef, C. “The Realities of Language Conversions”, IEEE Software, Nov. 2000, p.111 to 124.

[14] Sneed, H.M. “Human Cognition of Complex Thought Patterns – How much is our Perception of the Present determined by our Experience of the Past”, Keynote Address, 6th IWPC, IEEE Press, Ischia, Italy, June 1998.

[15] Tilley, Scott R. and Smith, Dennis B., “Perspectives on Legacy Systems Reengineering”, http://www.sei.cmu.edu/~reengineering/pubs/lsysree/, Reengineering Center, Software Engineering Institute, Carnegie Mellon University

[16] Shaw, Mary. "Architecture Issues in Software Reuse: It's Not Just the Functionality, It's the Packaging", in the proceedings of the IEEE Symposium on Software Reusability, April, 1995.

[17] Lee, Moon-Soo and Shin, Seok-Gyoo and Yang, Young-Jong, "The design and implementation of Enterprise JavaBean (EJB) wrapper for legacy system", IEEE International Conference on Systems, Man, and Cybernetics, October, 2001.

[18] Sneed, H.M., "Using XML to integrate existing software systems into the web", in the proceedings of the 26th International Computer Software and Applications Conference, August, 2002.

[19] Sneed, H.M., "Wrapping Legacy COBOL Programs Behind and XML-Interface", in the proceedings of the 8th Working Conference on Reverse Engineering, Stuttgart, Germany, October, 2001.

[20] Sneed, H.M., "Generation of stateless components from procedural programs for reuse in a distributed system", in the proceedings of the Fourth European Conference on Software Maintenance and Reengineering, March, 2000.

[21] Zdun, U., "Reengineering to the Web: a reference architecture", in the proceedings of the Sixth European Conference on Software Maintenance and Reengineering, March, 2002.

[22] Aversano, L., "Migrating Legacy Systems to the Web: an experience report", in the proceedings from the Fifth European Conference on Software Maintenance and Reengineering, March, 2001.

[23] Comella-Dorda, S. and Wallnau. K. and Seacord, R.C., and Robert, J., "A survey of black-box modernization approaches for information systems", in the proceedings from the International Conference on Software Maintenance, October, 2000.

[24] Sun Microsystems, "J2EE Connector Architecture Specification", http://java.sun.com/j2ee/connector

[25] OMG, "Corba Object Request Broker Architecture Specification", http://www.omg.org/corba

[26] Sun Microsystems, "Java Messaging Service Specification", http://java.sun.com/products/jms/

[27] IBM, "WebSphere MQ", http://www-306.ibm.com/software/integration/wmq/

[28] OASIS, "UDDI Spec Technical Committee Specification", http://uddi.org/pubs/uddi-v3.00-published-20020719.htm

[29] W3C, "Web Services Description Language (WSDL) 1.1", http://www.w3c.org/TR/wsdl, March 15th, 2001

[30] Livingston, D., "Advanced SOAP for Web Development". Prentice Hall, Upper Saddle River, 2002.

[31] W3C, "SOAP Version 1.2", http://www.w3c.org/2000/xp/Group/, June 26, 2002.

[32] Curbera, F. et al., "Unravelling the Web services web: an introduction to SOAP, WSDL, UDDI", Internet Computing, IEEE, March-April, 2002.

[33] W3C, "SOAP Messages with Attachments", http://www.w3c.org/TR/2000/NOTE-SOAP-attachments-20001211, December 11th, 2000.

[34] Nielson, Henrik et al., "WS-Attachments Specification", http://www-106.ibm.com/developerworks/webservices/library/ws-attach.html

[35] Microsoft, "Direct Internet Message Encapsulation (DIME)", http://www.gotdotnet.com/team/xml_wsspecs/dime/draft-nielson-dime-01.txt

[36] Roman E. et al., "Mastering Enterprise JavaBeans", John Wiley & Sons, New York, 2002.

[37] OASIS, "Relax NG Specification", http://www.oasis-open.org/committees/relax-ng/spec-22011203.html, December 3rd, 2001.

[38] Clark, James, "TREX – Tree Regular Expressions for XML", February 13th, 2001.

[39] Deem, Mike, "WSDL Extension for SOAP in DIME", http://www.gotdotnet.com/team/xml_wsspecs/dime/WSDL-Extension-for-DIME.htm, May 8th, 2002.

[40] Box, Don, "House of Web Services, The Continuing Challenge", http://msdn.microsoft.com/library/default.asp, June, 2002.

[41] OMG, "CORBA Language Mapping Specifications", http://www.omg.org/technology/documents/formal/corba_language_mapping_specs.htm, September 24, 2002.

[42] E. Gamma et. al. "Design Patterns: Elements of Reusable Object-Oriented Software", Addison Wesley, 1994.

[43] W3C, "XSL Transformations (XSLT)", November 16th, 1999.

[44] IBM, "HTTPR Specification", http://xml.coverpages.org/IBM-ws-httprspec.pdf, April 1st, 2002.

[45] Microsoft, IBM, VeriSign, "WS-Security", April 05, 2002, http://www-106.ibm.com/developerworks/library/ws-secure/

[46] W3C, "SOAP Security Extensions: Digital Signature", http://www.w3.org/TR/ SOAP-dsig/, February 6th, 2001.

[47] Franks J. et al., "HTTP Authentication: Basic and Digest Access Authentication", RFC 2617, June 1999.

[48] OASIS, "Security Assertion Markup Language", http://www.oasis-open.org/committees/security, 20th June, 2001.

[49] Helton, Rich, "Java Security Solutions", Wiley Publishing, Indianapolis, IN, 2002.

[50] OASIS, "The Business Transaction Protocol (BTP) 1.0", http://www.oasis-open/committees/business-transactions/, June, 2002

[51] IBM, "Business Process Execution Language", http://www-106.ibm.com/ developworks/library/ws-bpel, May 5th, 2003.

[52] Knutson, J et al., "Web Services for J2EE, Version 1.0", ftp://www-126.ibm.com/pub/jsr109/spec/1.0/websvcs-1_0-fr.pdf